VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

LOKALIZAČNÍ SYSTÉM ZALOŽENÝ NA OPTICKÝCH ZNAČKÁCH

FIDUCIAL MARKER-BASED LOCALIZATION SYSTEM

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. David Hála

AUTHOR

VEDOUCÍ PRÁCE — Ing. Petr Gábrlík, Ph.D.

SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. David Hála ***ID:*** 220889

Ročník: 2 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Lokalizační systém založený na optických značkách

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je vytvořit lokalizační systém pro mobilní robotiku pro vnitřní prostředí založený na kameře a optických značkách. Řešení bude využívat framework ROS2, již hotové nástroje pro detekci značek a systém Linux (Ubuntu).

1. Seznamte se s nástroji ROS2 a na systému Ubuntu se je naučte používat.
2. Seznamte se s principem lokalizačních značek (fiducial markers) a hotovými řešeními (ARTag, Aruco apod.).
3. Zvolený systém zprovozněte v ROS2 a otestujte praktické vlastnosti detekce.
4. Ve spolupráci s vedoucím promyslete, rozmístěte a zaměřte značky v části pracoviště UAMT.
5. Vytvořte SW řešení pro ROS2/Ubuntu umožňující lokalizaci mobilního robotu v rámci systému.
6. Ve spolupráci s vedoucím proveďte validaci řešení na reálné robotické platformě.

DOPORUČENÁ LITERATURA:

PYO, YoonSeok, HanCheol CHO, RyuWoon JUNG a TaeHoon LIM. ROS Robot Programming. Republic of Korea: ROBOTIS Co., 2017. ISBN 979-11-962307-1-5.

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 15.5.2024

Vedoucí práce: Ing. Petr Gábrlík, Ph.D.

doc. Ing. Petr Fiedler, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Tato práce se zabývá lokalizací robota v prostoru za pomocí optických značek. V první části práce je čtenáři nastíněna problematika a úskalí tohoto projektu a jsou vysvětleny veškeré důležité souvislosti, ať už matematické či fyzikální. Druhá část je pak zaměřena na popis postupu, který byl zvolen při návrhu programu a myšlenky, které k jeho dokončení vedly.

KLÍČOVÁ SLOVA

ROS, ROS2, transformace, lokalizace, robot, quaterniony, tf2

ABSTRACT

This work deals with localization of the robot in space using optical markers. In the first part of the thesis, the reader is introduced to the problems and pitfalls of this project and all the important connections, both mathematical and physical, are explained. The second part then focuses on the description of the procedure that was chosen to design the program and the ideas that led to its completion.

KEYWORDS

ROS, ROS2, transformation, localization, robot, quaternion, tf2

Vysázeno pomocí balíčku `thesis` verze 4.09; `https://latex.fekt.vut.cz/`

HÁLA, David. Lokalizační systém založený na optických značkách [online]. Brno, 2024 [cit. 2024-05-15]. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/160059. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Petr Gábrlík.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. David Hála
VUT ID autora:	220889
Typ práce:	Diplomová práce
Akademický rok:	2023/24
Téma závěrečné práce:	Lokalizace na základě optických značek

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing.Petru Gábrlíkovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci, které vedly k jejímu dokončení.

Obsah

Úvod **11**

1 Teoretická část studentské práce **12**
1.1 ROS . . . 12
1.1.1 Topic . . . 12
1.1.2 Service . . . 13
1.2 Transformace . . . 14
1.2.1 Pohyb a rotace . . . 15
1.2.2 Quaterniony . . . 17
1.3 Kamerový vstup . . . 18
1.3.1 CCD vs CMOS . . . 19
1.4 Kalibrace kamery . . . 20
1.5 Detekce . . . 22
1.5.1 Princip detekce . . . 22
1.6 Statistické veličiny . . . 25
1.7 Kalmanův filtr . . . 27

2 Výsledky studentské práce **30**
2.1 Využité knihovny . . . 31
2.1.1 Aruco ROS . . . 31
2.1.2 tf2 . . . 32
2.1.3 UsbCam . . . 33
2.1.4 Image_pipeline . . . 33
2.2 Preparace a nastavení . . . 33
2.2.1 Prostředí . . . 33
2.2.2 Kalibrace kamery . . . 34
2.2.3 Statické transformace . . . 35
2.3 Průzkum knihovny Aruco_ROS . . . 36
2.3.1 Soubor CmakeLists . . . 39
2.3.2 Launch soubor . . . 39
2.4 Simulace . . . 40
2.5 Lokalizační balíček Aruco ROS locator . . . 42
2.5.1 Výběr jazyka . . . 43
2.5.2 Měření přesnosti . . . 44
2.5.3 Asynchronní programování . . . 48
2.5.4 Popis programu . . . 49
2.5.5 Celkové shrnutí . . . 58

2.5.6 Testování 59
2.5.7 Postup instalace 69

Závěr **72**

Literatura **74**

Seznam obrázků

1.1 ROS Topic . . . 13
1.2 Příklad ROS Topicu - Turtlesim . . . 13
1.3 ROS Service . . . 14
1.4 1D rotace . . . 15
1.5 2D rotace . . . 16
1.6 Rotace quaternionů . . . 18
1.7 Kamerové filtry . . . 19
1.8 CCD vs CMOS . . . 20
1.9 Aruco marker . . . 22
1.10 Příklad histogramu . . . 23
1.11 Příklad prahování . . . 23
1.12 Detekce pomocí Cannyho detektoru . . . 24
2.1 Transformace robota . . . 30
2.2 Ukázka instalačního souboru ArucoRos knihovny . . . 31
2.3 Kalibrace kamery . . . 35
2.4 Graf aplikace . . . 38
2.5 Simulace transformačního stromu . . . 41
2.6 Výsledná simulace . . . 42
2.7 Kompilace C++ vs Python . . . 43
2.8 Pokus měření přesnosti . . . 46
2.9 Výsledný Excel tabulka a hodnoty . . . 48
2.10 Synchronní programování . . . 49
2.11 Asynchronní programování . . . 49
2.12 Mód nearest . . . 52
2.13 Mód average . . . 53
2.14 Průměrování rotací . . . 55
2.15 Lokalizace bez rotační matice vs s maticí . . . 57
2.16 Graf lokalizace . . . 59
2.17 Graf lokalizace . . . 60
2.18 Schéma pokusu . . . 61
2.19 Robot Loki . . . 63
2.20 Rozložení místnosti . . . 64
2.21 Průběh měření . . . 66
2.22 Průběh finálního měření . . . 68
2.23 Průběh finálního měření . . . 69

Seznam výpisů

2.1 Příklad vytváření balíčku. 34
2.2 Příklad kompilace. 34
2.3 Příklad spouštění kamery pro kalibraci. 34
2.4 Příklad kalibrace kamery. 34
2.5 Příklad spouštění kamery po kalibraci. 35
2.6 Příklad spouštění kamery po kalibraci. 37
2.7 Příklad vypisování výsledné transformace. 38
2.8 Příklad CmakeLists kódu 39
2.9 Příklad definice proměnné 40
2.10 Příklad definice pomocí příkazového řádku 40
2.11 Příklad definice pomocí launch file 40
2.12 Příklad spouštění simulačního programu. 41
2.13 Příklad async funkce v Pythonu 49
2.14 Příklad spuštění lokátoru 50
2.15 Příklad tvorby subscription 51
2.16 Výpočet vazby 55
2.17 Přepočet vektoru 56
2.18 Tvorba lokální vazby 57
2.19 Tvorba lokální vazby 58
2.20 Vazby v setup.txt 60
2.21 Vazby v setup.txt 64
2.22 Vazby v setup.txt 65
2.23 Chyba instalace OpenCV 70
2.24 Chyba instalace OpenCV 70
2.25 Instalace knihoven 70
2.26 Instalace knihoven 71

Úvod

Orientace robota v prostoru je nedílnou součástí automatizace v odvětví robotiky. Tento obor s sebou přináší spoustu komplikací a nástrah, které bylo nutné vyřešit a to nikoli pouze pro účely této diplomové práce, ale také ve prospěch mého osobního rozvoje jakožto absolventa fakulty elektrotechniky. Mým úkolem jakožto autora této práce je seznámit čtenáře s podstatou problematiky orientace v prostoru a vysvětlit její zákonitosti. Na základě tohoto teoretického rozboru bylo následně nutné navrhnout a implementovat řešení, které by co nejvíce zmenšilo odchylku a tudíž zlepšilo lokalizaci v již existujícím systému, aplikovaném na reálném robotu. Toto téma jsem si zvolil, neboť je úzce spjato s obory, které jsou pro mne nanejvýš zajímavé a především přínosné. Těmito obory jsou robotika a počítačové vidění. Zvolil jsem si jej ale také proto, že jeho výstupem je program s reálnou aplikací nejen pro budoucí generace studentů, ale inženýrů obecně. Jako autor mám tak jedinečnou možnost přispět alespoň touto formou k vývoji robotiky jako takové a ulehčit práci všem, co se tímto oborem zabývají.

Praktická část práce byla vypracovávána na platformě ROS, jež je celosvětově používaný nástroj pro vývoj aplikací v robotice. Pro detekci pak byla použita knihovna Aruco z důvodu, že se jedná o široce používanou knihovnu. Díky obsáhlosti dokumentace, kterou knihovna nabízí, bylo výrazně přispěno k pochopení celého tématu.

Po přečtení teoretického rozboru by měl být čtenář seznámen s rozhraním ROS, vědět jak fungují transformace a co jsou to Quaterniony. Dále bude popsáno, jak funguje kamera, k čemu slouží její kalibrace a nebo jak probíhá samontá detekce značek. Čtenář by tak nadéle neměl mít žádný problém s pochopením postupu výoje výsledné aplikace a matematikou, co stojí za její funkčností.

1 Teoretická část studentské práce

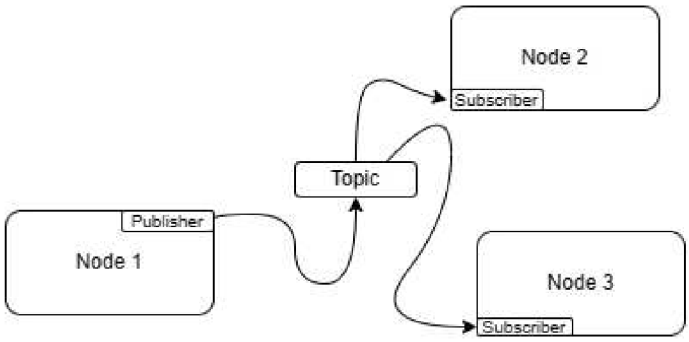
1.1 ROS

ROS (Robot Operating System) je open-source framework (balíček nástrojů), který vznikl v roce 2007 na Stanfordské univerzitě. [2] Od té doby bylo vytvořeno velké množství aplikací, při kterých bylo možné a taky praktické ROS využítt. Dnes je již standartně využívaným nástrojem pro vývoj software robota. Při tvorbě takiového software, je potřeba zaručit, aby komponenty, jako například kamery, senzory, motory a jiné, spolupracovaly za účelem splnění požadavku operačního systému. Pro každou komponentu je standartně vytvářen samostatný kód, který je v ROS terminologii označován jako „Node“ (v práci dále jako „uzel"). Ve výsledné aplikaci pak tyto uzly běží paralelně a jsou mezi nimi vytvářeny komunikační kanály pro výměnu informací. Komunikační kanály existují dva hlavní typy – topic a service. [2]

1.1.1 Topic

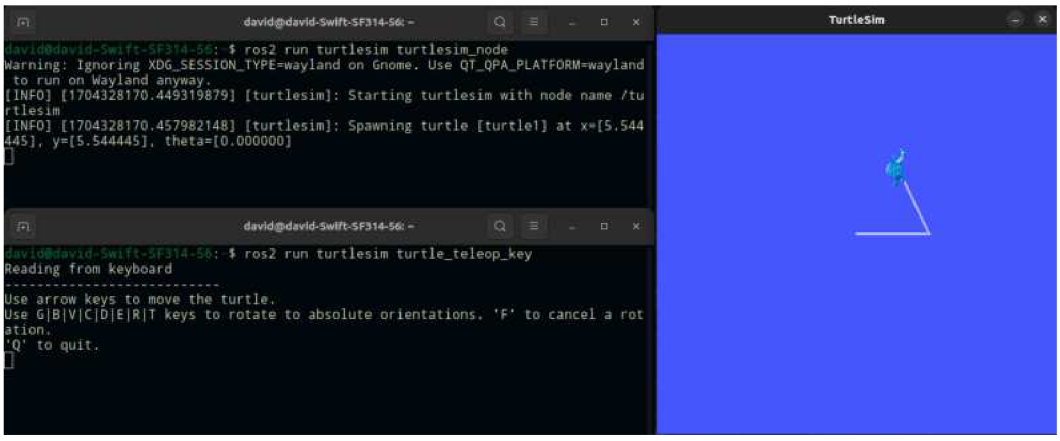
Topic je typ komunikace, kdy na jedné straně (v jednom uzlu) je vytvořen odesílatel a na straně druhé adresát. Jedná se o výměnu informací mnoha k mnohým. [2] [1] To znamená, že do jednoho topicu může přispívat vícero odesílatelů a zároveň být čten vícero adresáty. Tohoto typu je využíváno pouze k přenosu zpráv jedním směrem. Obousměrná komunikace není možná. Varianta je tak vhodná, pakliže vyloženě není oboustranná výměna informací po kanále vyžadována. Topic jako typ komunikace je využíván pro rychlou publikaci periodicky obdržovaných informací. Dle této charakteristiky je jasné, že topicy jsou vhodné k využití hlavně pro odesílání informací, obdržených ze senzorů, jiným uzlům, kde jsou nad daty prováděny další operace. V případě této diplomové práce se například může jednat o kamerový obraz, který je posílán v podobě matice a je posílán jinému uzlu k detekci Dále se jedná o ryze asynchronní výměnu a není tedy není potřeba jakákoliv časová synchronizace komunikujících stran. [1] Zpráva je jednoduše adresátům dostupná, dokud nevyprší její platnost (nejsou obdrženy další data).

Topic je možné ukázat na předem předpřipraveném balíčku turtlesim, který je součástí prostředí ROS. Prvně je spuštěn uzel `turtlesim_node`, jehož podstatou je vykreslení želvy v prázdném pozadí okna na obrazovce. V tento moment nicméně ještě není možné se želvou jakýmkoliv způsobem interagovat. K tomu je navíc potřeba spustit uzel „teleop-key“. Teleop key je uzel, běžící paralelně s `turtlesim_node` a skrze který, jsou obstarávány informace o stisknutí klávesy na



Obr. 1.1: ROS Topic

klávesnici. V důsledku přijmutí informace od teleop-key skrze jednosměrný komunikační kanál, je se želvou v `turtlesim_node` pohybováno. V tomto případě není nutné vracet teleop-key žádnou odpověď. Naopak uzel slouží čistě k dodávání informace do `turtlesim_node` od fyzického komponentu, kterým je v tomto případě klávesnice a tím je jeho podstata naplněna. Uzlem `turtlesim_node` je zase této informace využíváno k vyvolání akce pohybu. Tato akce tak nemusí být volána a kontrolována při každém cyklu, ale je volána pouze tehdy, kdy jsou skkrze teleop-key doručena data.

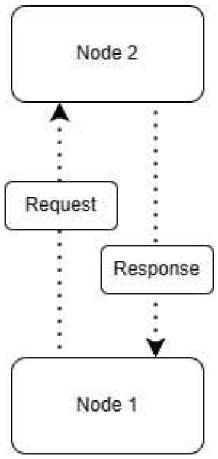


Obr. 1.2: Příklad ROS Topicu - Turtlesim

1.1.2 Service

Service je založen na principu klient-server komunikace. [1] [2] Na jedné straně je tedy vytvořen server a na straně druhé pak klient, který je přihlášen k odběru jeho dat. Opět zde platí, že klientů může být několik. Server v tomto případě ale, může být pouze jeden. Důležité je podotknout, že v tomto případě se jedná o obousměrný komunikační kanál. [1] [2] Jeden z klientů je tedy schopen si vyžádat informace

z běžícího serveru, od kterého je zpět poslána odpověď na požadavek. Tento typ komunikace je na rozdíl od topiců synchronní.[1] Je tedy nutné strany časově synchronizovat, aby byly v daný okamžik obě strany připraveny na následující započatí komunikace. Service je vhodný k využití tam, kde je obousměrná komunikace nutná a není potřeba se na server dotazovat v tak krátkých intervalech, jako je tomu třeba při použití topicu. [2] Zejména tomu tak může být při dotazování se na určité parametry, databáze či její části, kde je použití komunikace typu service vhodné. [1]



Obr. 1.3: ROS Service

Pro příklad použití service je opět možné využití, již známého balíčkku turtlesim, který byl popsán v předchozí podkapitole. Za Želvou, pohybující se po prázdném pozadí, je vykreslována trajektorie jejího pohybu pohybu v podobě přímek různých barev. Standartně je barva této přímky bílá. Je ale možné barvu měnit, pomocí dotazu na service `/turtle1/set_pen`. Předpoklad je takový, že uživatel nebude chtít barvu trajektorie periodicky měnit každých několik setin sekundy, ale spíše je barva nastavena jednou, maximálně párkrát v průběhu celého programu, a tedy je použití service vhodné.

1.2 Transformace

Problematika transformace je nanejvýš důležitým tématem této diplomové práce. Pro jednoduchost si prvně představme, že v 3dimenzionálním prostoru je sledován nekonečně malý bod. Příkladem tohoto bodu může být střed fotbalového míče. Zde je problém transformace jednoduchý.Nekonečně malý bod v prostoru je . Pohyb objektu je tedy patřičně omezen svými rozměry a má ve výsledku pouze 3 stupně volnosti – pohyb v ose x, pohyb v ose y a pohyb v ose z. V tomto případě je totiž

předpokládáno, že u nekonečně malého objektu můžou být rotace zanedbány. Jeho transformaci je možné popsat pomocí následující rovnicí:

$$T_v \cdot P = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{bmatrix} = p + v, \tag{1.1}$$

kde p značí pozici objektu a v vyjadřuje vektor pohybu.

V reálné aplikaci samozřejmě není možné pracovat s matematickými modely nekonečně malého bodu a je tedy nutné do rovnice zahrnout i rotaci objektu v každé z os. Stupeň volnosti objektu v 3dimenzionálním prostoru je tedy roven šesti, což nadále komplikuje problematiku transformace. Pro výpočet takové transformace mohou být mimo jiné, využívány Quaterniony.

1.2.1 Pohyb a rotace

Způsob popisu rotace v 3dimenzionálním prostoru není tedy zcela tak triviální, jak se může zdát. Reálná čísla jsou v tomto případě nedostatečná a je nutné využít tři různé imaginární proměnné pro vyřešení daného problému. Aby bylo možné lépe problematiku pochopit, začněme nejdřív v 1dimenzionálním prostoru. Tedy v prostoru, který je definován pouze reálnými čísly. Řekněme, že existuje na přímce P bod B, jehož souřadnice je 0. Pokud je k této pozici přičteno číslo 2, pak bod B je po přímce P posunut o 2 kroky doprava. Reálná čísla tedy je možné chápat jako posuv bodu po přímce. Rotace v tomto prostoru jsou jednoduché, jelikož jediná prostorem povolená rotace je převrácení bodu o 180°, kdy se ze souřadnice 2 stane -2.

Obr. 1.4: 1D rotace

Jelikož byla reálná čísla definována jako posuv po přímce, není s nimi možné popsat pohyb v jakémkoliv prostoru s dimenzí větší jak 1. Z tohoto důvodu je nutné, aby byl představen koncept imaginárních čísel. Imaginární číslo je značeno písmenem i a jeho hodnota je:

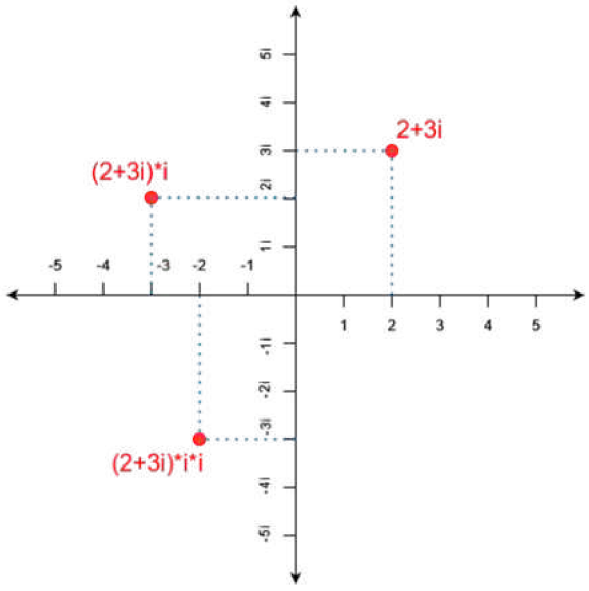
$$i = \sqrt{-1} \tag{1.2}$$

Definujme imaginární číslo, jako číslo operující v 2dimenzionálním prostoru. Toto číslo popisuje pohyb bodu v ose, která je kolmá na osu reálných čísel, tedy na osu posuvu. Mějme opět bod B, jehož poloha je definována na osách X a Y, se souřadnicemi 0,0 (pohyb v ose x, pohyb v ose y). Tento bod je následně posunut do souřadnic 2,3, což je možné napsat také jako:

$$2 + 3i \tag{1.3}$$

Jelikož se pohybujeme ve 2 dimenzích, prostor je rovinou, na které je možné definovat rotace. Jak již bylo řečeno, osa Y, na které je pohyb definován pomocí i, je kolmá na osu X, tedy imaginární i představuje rotaci 90°. Řekněme, že dané souřadnice {2,3} je nutné orotovat o i, tedy:

$$\begin{aligned} (2 + 3i) \cdot i &= 2i + 3 \cdot i^2 \\ i^2 &= \sqrt{-1}^2 = -1 \\ (2 + 3i) \cdot i &= -3 + 2i \end{aligned} \tag{1.4}$$



Obr. 1.5: 2D rotace

Když byl nyní popsán a pochopen koncept rotace v 2dimenzionálním prostoru, je možné se přesunout na 3dimenzionální prostor. Bod v tomto prostoru je samozřejmě definován pomocí 3 souřadnic – posuv v ose X, Y a Z. Logicky by se dalo uvažovat, že pokud pro popis rotace v 2dimenzionálního prostoru postačilo 2dimenzionální číslo (Reálná a imaginární složka), je možné tuto logiku aplikovat i na 3dimenzionální prostor a použít 3dimenzionální číslo. Nicméně na rozdíl od 2dimenzionálního

prostoru, je možná rotace ve všech osách pohybu. Zatímco u 2-D prostoru byl počet prostorem povolených rotací o 1 menší než počet dimenzí, zde tomu tak není. Pro popis rotace v tomto prostoru je tak potřeba využít 4dimenzionální číslo – Quaternion. [4]

1.2.2 Quaterniony

Jak již bylo řečeno v předchozí kapitole, quaterniony jsou 4dimenzionální čísla, popisující rotaci v 3dimenzionálním prostoru. [3] [4] [5] Quaternion je možné definovat například jako:

$$q = 2 + 3i + 4j + 5k \tag{1.5}$$

Každý quaternion je složen vždy z reálné části a 3 imaginárních složek, kterým je přezdíváno vektorová část. [3] [5] Pro vektorovou část a quaterniony jako takové je nutné nadále specifikovat pravidla součinu:

-	**1**	**i**	**j**	**k**
1	1	i	j	k
i	i	-1	-1	-j
j	j	-k	-1	i
k	k	j	-i	-1

[3] [5] Tato pravidla jsou jádrem celé rotace objektu. Představme si prostor, definovaný 3 osami X,Y,Z. Dané osy mají vždy rozmezí od <-1;1> × příslušející imaginární proměnná:

$$\begin{aligned} X &\in C < -i; i > \\ Y &\in C < -j; j > \\ Z &\in C < -k; k > \end{aligned} \tag{1.6}$$

Celý prostor tedy tvoří quaternion, který je ve tvaru:

$$v = 0 + i + j + k \tag{1.7}$$

Pokud nyní bude potřeba celý objekt otočit o i, dosáhneme toho vytvořením dalšího quaternionu, určující směr a velikost rotace.

$$q = 0 + i + 0j + 0k \tag{1.8}$$

Zde se opět dostáváme k pravidlům součinu, které byly specifikovány výše v textu. Je možné si povšimnout, že jestliže je každá jedna proměnná ve vektorová části vynásobena i, pak nastává rotace celého prostoru ve 2 směrech. Pro eliminaci rotace v

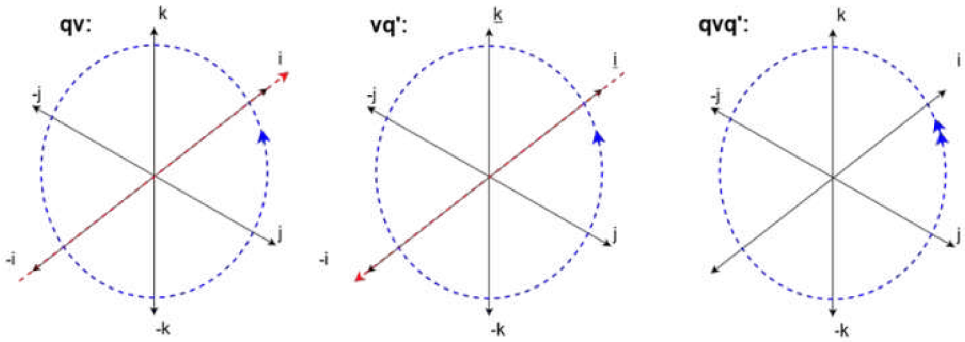
jednom směru, je vytvořen konjungovaný quaternion, jehož číslené hodnoty zůstávají stejné, ale vektorová část je záporná. [3] [4]

$$q' = 0 - i - 0j - 0k \tag{1.9}$$

Směr rotace konjungovaného quaternionu je v ose X opačná, zatímco rotace ve směru ostatních os zůstávají nezměněny. Následně využijeme konjunkci pro eliminaci rotace:

$$v = qvq', \tag{1.10}$$

kde q je quaternion, určující rotaci, q‘ je jeho konjunkce a v je čistý quaternion (s nulovou reálnou složkou), určující prostor.



Obr. 1.6: Rotace quaternionů

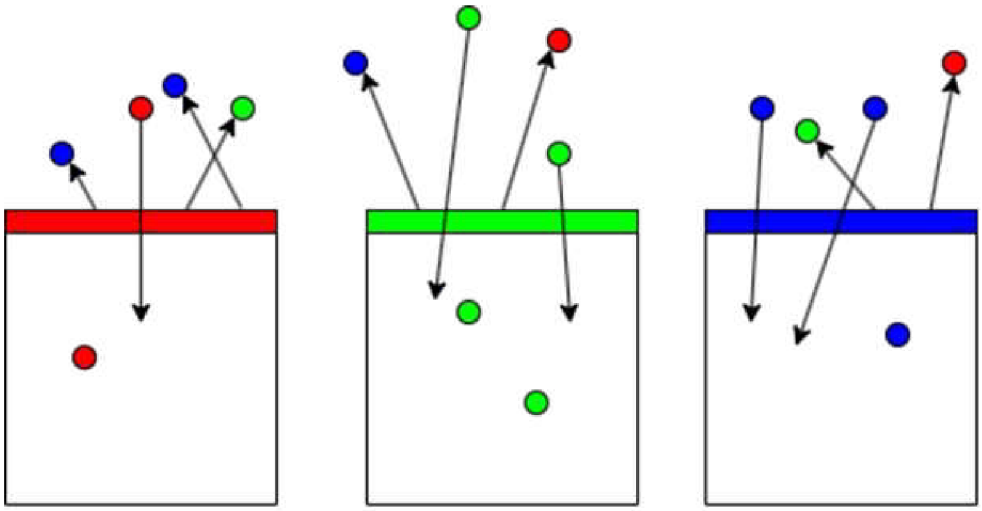
Problémem je, že takto definovaná rotace zdvojnásobí úhel, o který se má výsledný objekt otočit. Je tak nutné s tímto faktem počítat a požadovaný úhel podělit dvěmi před samotnou operací s quaterniony. [3] Výsledný quaternion tak musí být ve tvaru:

$$q = \cos(\theta) + \sin(\frac{\theta}{2}) \cdot (xi + yj + zk) \tag{1.11}$$

1.3 Kamerový vstup

Před zpracováním jakékoliv informace z obrazu je prvně nutné samotný obraz přijmout. K tomu byla v rámci diplomové práce využita webkamera na notebooku a full HD kamera zapůjčena vedoucím práce. Obě tyto kamery fungují ale na naprosto stejném principu. Kovové šasi notebooku disponuje otvorem, do kterého je přirozeně přiváděno světlo skrze vložené čočky. Čočka je vpodstatě pouze sklo, které je zbroušeno na požadovaný tvar a plní podobnou funkci, jako plní skla v dioptrických brýlích. Tímto efektem je korekce ohniskové vzdálenosti. [11] Ohnisková vzdálenost je taková vzdálenost, kde se protínají paprsky světla, vystupující z čočky. Tato proměnná určuje nejen, na jaký objekt je kamerový obraz zaostřen, ale také jaký úhel

zorného pole, je schopna kamera zachytit. [10] Světlo je poté přivedeno na matici světlo-citlivých senzorů, které slouží k převodu množství dopadajících fotonů na elektrický signál. Úroveň signálu produkovaného senzorem je změřena a z jeho hodnoty je odvozena hodnota, reprezentující jasovou úroveň obrazového bodu (pixelu). [11] Takto popsaná kamera by ale byla schopna generovat pouze černobílý obraz, jelikož každý pixel je určen pouze jednou hodnotou jasové úrovně, z které není možné získat informaci o barvě. Každá barva je ve skutečnosti složena ze tří základních barev – R (červená), G (zelená), B (modrá). Pro snímání barevného obrazu je tak každý pixel reprezentován třemi senzory, kdy je každý z této trojice senzorů určen právě ke snímání jedné složky RGB obrazu. Aby toto bylo možné zaručit, je před senzor vložen filtr, který vždy propustí pouze světlo s požadovanou vlnovou délkou (požadované barvy). [11]

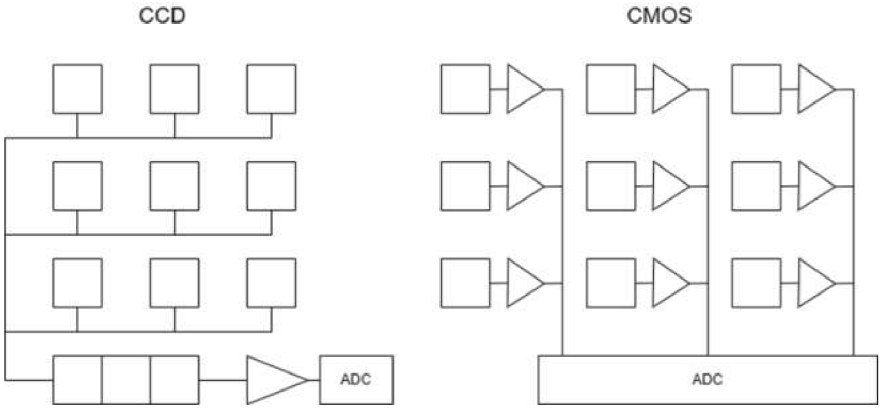


Obr. 1.7: Kamerové filtry

1.3.1 CCD vs CMOS

CCD i CMOS jsou typy světlo-citlivých senzorů, široce používaných pro aplikaci v digitálních kamerách. U obou těchto typů, je využíván tzv. fotoelektrický jev, k převodu světla, dopadajícího na senzory na elektrický signál. [14] Zde nicméně podobnost těchto dvou zařízení končí. Zatímco CCD produkují ostřejší obraz s menším množství šumu, Pomocí CMOS je zase uživateli nabízena nižší spotřeba, vyšší rychlost a menší rozměry. CCD obsahuje jeden zesilovač, skrze který jsou náboje ze senzorů jeden po druhém posouvány a zesilovány zvlášť před samotnou konverzí na digitální hodnotu (číslo). [12] CMOS obsahuje pro každý jeden senzor zvlášť MOSFET transistory, zastupující zesilovač a plnící stejnou funkci. [13] Tedy v překladu je zesilován každý pixel vlastním zesilovačem a je tedy možné, aby bylo v jeden mo-

ment zpracováno mnohem více pixelů, než je tomu u CCD, který je v tomto ohledu značně limitován. [13] Je tedy jasné, že CMOS bude znatelně rychlejší. Mimo tento fakt je další velkou výhodou celková spotřeba. Ta může v některých případech nabývat až stokrát vyšší úspory, než je tomu u konkurenčího senzoru. [14] Tyto pozitivní vlastnosti jsou ale bohužel vykoupeny určitými nevýhodami. Senzory CMOS je produkován méně kvalitní obraz s většími hodnotami šumu, které jsou zapříčiněny vyšším počtem elektronických součástek. [13] Nicméně pro aplikace počítačového vidění a rozpoznávání, je i přes tuto nevýhodu vhodné použití výhradně kamer, využívajících technologii CMOS. Zatímco zmiňované rozdíly v kvalitě obrazu nejsou nikterak problematické, rychlost zpracování obrazu je naprosto klíčová k tvorbě plynulého obrazu vstupu.



Obr. 1.8: CCD vs CMOS

1.4 Kalibrace kamery

Kalibrace je metoda zjišťování geometrických charakteristik kamery, měřením tzv. vnitřních a vnějších parametrů. [15] [16] Vnější parametry se zabývají lokalizací a orientací kamery jako celku, zatímco vnitřní parametry se zabývají jednotlivými prvky kamery. Těmi jsou například ohnisková vzdálenost, zorné pole a tak podobně. [15] [16] Správná kalibrace je obzvláště důležitá, pokud je nutné pomocí obrazu měřit vzdálenost od objektu nebo jeho rozměry jako takové. Při jejím správném provedení, existuje přesná vazba mezi bodem ve 3D prostoru a jeho projekcí do 2D roviny (obrazu). [16]

Kalibrace se většinou sestává z několika málo kroků. Prvně je určen kalibrační objekt. Tím může být kniha, krabice, nebo . Je ale nutné znát přesné rozměry L_x a L_y tohoto objektu. Dalším krokem je postavení objektu od objektivu, v přesně definované vzdálenosti L_z, kolmo ke kameře. V tento moment je často pořízeno foto

objektu a jsou změřeny jeho rozměry L_{xp} a L_{yp} v pixelech. [16] Z těchto hodnot jsou následně vypočítány tyto parametry:

$$\begin{aligned} f_x &= \frac{L_{xp}}{L_x} \cdot L_z \\ f_y &= \frac{L_{yp}}{L_y} \cdot L_z \end{aligned} \tag{1.12}$$

Pomocí výše uvedených hodnot je možné následně sestavit tzv. matici vnitřních parametrů:

$$K_{int} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{1.13}$$

kde a c_x a c_y jsou střed obrazu a tedy se rovnají polovině délky obrazu v pixelech a s_x je tzv. natočení roviny (skew). [15] Abychom mohli kalibraci dokončit, je nutné ještě definovat matici vnějších parametrů. Správné nastavení hodnot této matice je obtížnější, než se může zprvu zdát. Samotná matice je definována jako:

$$L_{ext} = \begin{bmatrix} R \\ t \end{bmatrix}, \tag{1.14}$$

kde t je translace a R je rotace kamery v 3D prostoru. [16] Zjistit translaci kamery od sledovaného objektu není nijak problematické a v podstatě byla tato hodnota již získána, při naplňování matice K_{int}. Rotace kamery je ale mnohem složitější. I toto je samozřejmě možné určitým způsobem změřit, je ale velice obtížné dosáhnout nulové odchylky. Při aplikacích, jako je tato naštěstí nepředstavuje určitá odchylka problém, který by vedl k selhání funkce.

Po výpočtu obou těchto matic je kalibrace hotová a mezi pixely v obraze a bodem v 3D rovině nyní existuje statická vazba ve tvaru:

$$w \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \times K_{ext} \times K_{int}, \tag{1.15}$$

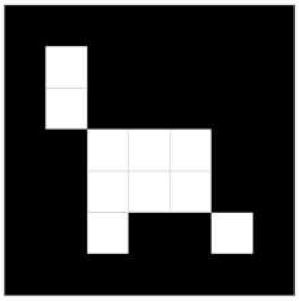
kde w značí váhu, velká písmena souřadnice ve 3D rovině a malá značí pixely v obraze. [15]

Kalibrace pomocí šashovnice

V případě této diplomové práce, byla ke kalibraci použita tzv. kalibrační šachovnice. Ta byla vytištěna na list papíru a její čtverce byly patřičně přeměřeny. Tedy přesné rozměry jednotlivých polí a tudíž i celé šachovnice byly známy. V tomto případě není nutné počítat přesné vzdálenosti od kamery.

1.5 Detekce

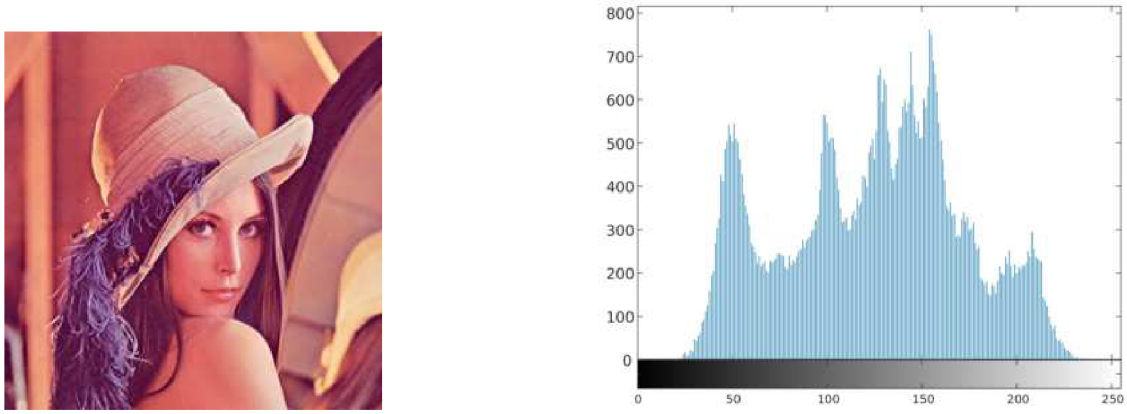
Pro tvorbu této diplomové práce byla využita knihovna ArucoRos od firmy pal-robotics. Ta slouží jako implementace knihovny Aruco, které bylo vyvinuto seskupením Applicaciones de la Visión Artifical do prostředí ROS. Aruco je open-source knihovna, která slouží k detekci a rozpoznávání optických značek (tzv. Aruco markerů) pomocí kamery. Aruco markery jsou proprietární značky, připomínající QR kód, které v sobě nesou číselnou informaci. Na základě velikosti značky je možné generovat čísla od 0 až po 1023. Tedy maximálně značka obsahuje 10 bitů informace. [18]



Obr. 1.9: Aruco marker

1.5.1 Princip detekce

Samotná detekce těchto značek je pak prováděna na základě několika málo kroků. Jako první je na obraz aplikováno adaptivní prahování (adaptive thresholding). [22] Prahování je metoda, kterou lze jednoduše pochopit, pokud si zobrazíme obrazový histogram. Histogram je graf, popisující četnost. V tomto případě se jedná o četnost jasových hodnot pixelů v obrázku.



Obr. 1.10: Příklad histogramu

Pro následující příklad byl obraz převeden do černobíle podoby. Nyní je určena hodnota (práh), která reprezentuje limitu jasových hodnot. Například budou brány v potaz pouze takové pixely, jejichž jasová hodnota je vyšší, než 150, čímž vznikne obraz, který zvýrazňuje kontury v nezávislosti na kvalitě osvětlení. Pokud ale obraz obsahuje nekonzistentní osvětlení a stíny, není možné na celý obraz použít jednotnou hodnotu a je nutné jej rozdělit na určité kvadranty a práh adaptovat na úrovně osvětlení těchto jednotlivých kvadrantů.

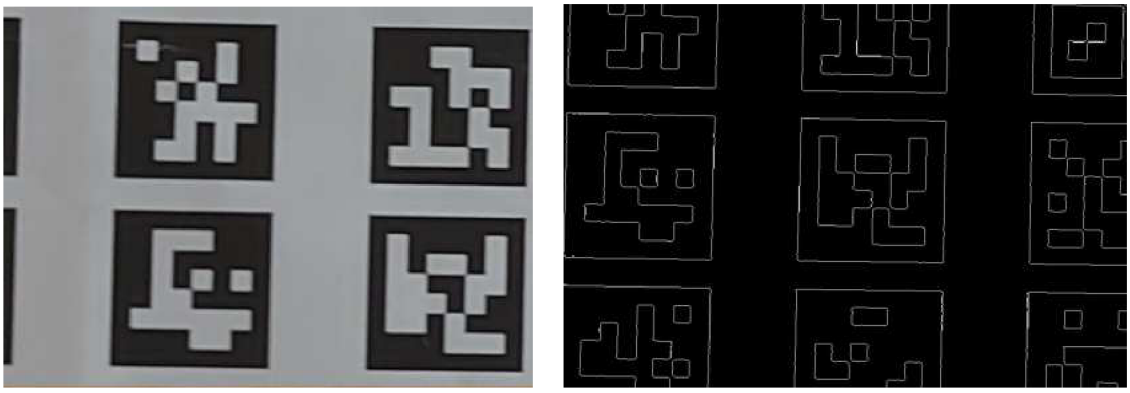
Obr. 1.11: Příklad prahování

Nyní se v takto zpracovaném obrázku naleznou kontury (hrany). [22] Toho je možné dosáhnout pomocí tzv. filtrů. Dokumentace ([22]) nespecifikuje přesně, jaký filtr je v algoritmu použit, ale pro vysvětlení podstaty byl algoritmus zreplikován za pomocí Cannyho detektoru. V kapitole 1.3 bylo zmíněno, že každý pixel je definován jasovými úrovněmi jeho složek. Jelikož byl ale z výsledku prahování obdržen černobílý obraz, figuruje zde pouze jedna jeho složka. Obrázek je tedy možné vyjádřit

pomocí jedné matice:

$$Img = \begin{bmatrix} 0 & ... & 255 \\ : & ... & : \\ 255 & ... & 0 \end{bmatrix} \tag{1.16}$$

Canny detektor jako první obraz vyhladí, s pomocí Gaussovského filtru. [23] Filtr je opět ve formě matice, jehož hodnoty jsou reprezentací Gaussovy funkce. Filtr je následně s obrazem konvulovén, což ve výsledku znamená, že jsou hodnoty filtru otočeny a matice je následně posouvána po obraze, kde jsou jejich vzájemné hodnoty násobeny. Po této operaci je vypočtena derivace, čímž je získán gradient, velikost a směr gradientu v obraze v osách X i Y. Jako poslední je obraz prahován a jsou zachovány pouze ty pixely, jejichž velikost gradientu je nad určitou hranicí, kterou je možné v kódu specifikovat. [23] Výsledek po aplikaci Cannyho detektoru je vidět na obrázku níže.



Obr. 1.12: Detekce pomocí Cannyho detektoru

Obrázek však většinou neobsahuje pouze značky, ale všemožné objekty a jiný informační šum. Informačním šumem je myšlena každá hrana objektu, která byla zaznamenána, ale není předmětem detekce. Algoritmus Aruco tedy vyfiltruje hrany, které mají nízký počet pixelů, čímž je výrazně snížen počet možných hran v obraze. [22]

Pomocí polynomiální aproximace kontur jsou zjištěny tvary veškerých zbývajících hran a jsou následně ponechány pouze ty, vykazující konkávní charakter s přesně čtyřmi rohy – obdélníky nebo čtverce. Rohy jsou poté seřazeny v proti směru hodinových ručiček a jsou odstraněny ty obdélníky, jejichž vzájemná poloha je příliš krátká. Adaptivní prahování totiž většinou detekuje vnitřní i vnější hranu značky. V tomto případě je odstraněna vnitřní hrana a vnější ponechána. Značky v prostoru nemusí být a často ani nejsou otočeny přesně kolmo směrem ke kameře. Je tedy nutné použít tzv. homografii. [22] Homografie je metoda, kterou lze otáčet objekty v

obraze. [20] V kapitole 1.2.2 byl vysvětlen koncept natáčení objektu ve 3D prostoru. Zde to funguje velice podobně. Existuje tedy obraz s maticí pixelů P a transformační matice H, obsahující vnitřrní a vnější paramtery kamery. Výsledný natočený obraz P_{rot} se vypočte jako:

$$P_{rot} = H \times P, \tag{1.17}$$

kde H je transformační matice a P je obraz objektu. K sestrojení transformační matice H je však nutné kameru, kterou obraz pořizujeme zkalibrovat. [20] Po natočení značky je vypočten nový práh pomocí Otsu metody, která se snaží maximalizovat meziskupinovou varianci. [22] Otsu rozdělí pomocí jasových hodnot pixely na dvě skupiny - pozadí a popředí. Následně je práh zvyšován či snižován tak, aby mezi těmito skupinami vznikaly, co největší rozdíly. Toho je docíleno pomocí rovnice:

$$\sigma_B^2 = W_b W_f \cdot (\mu_b - \mu_f)^2, \tag{1.18}$$

kde W_b a W_f je počet pixelů ve skupině a μ_b nebo μ_f je střední hodnota intenzity. [21]

Nyní již algoritmus lokalizoval umístění značek v obraze, pomocí filtru bylo zjištěno, kde jsou ohraničeny a s využitím homografie, jsou značky natočeny kolmo ke kameře. Posledním krokem při detekci je tedy samotné rozpoznání dat. Značka je rozdělena na 6x6 mřížku s daty o velikosti mřížky 5x5. Podle barvy jsou poté rozlišovány bity na hodnoty jedna a nula. Tedy je-li pole bílé, znamená to, že příslušný bit je roven jedné a naopak. [22]

1.6 Statistické veličiny

Statistické veličiny jsou velkým tématem této diplomové práce. Výsledný program obsahuje skript na výpočet určitých paramterů, skrze měření značek Aruco v předem definovaných vzdálenostech. Přesněji řečeno se hlavně jedná o mean nebo také střední hodnota, std neboli rozptyl za předpokladu normálního rozložení, což obrazový šum splňuje a RMSE, což je zkratka označující střední kvadraitckou chybu.

Střední hodnota není nutná zdlouhavě vysvětlovat, jedná se o hodnotu, která leží přesně na středu rozsahu mezi maximem a minimem sledované veličiny. Za tímto účelem existuje v programu proměnná, která sčítá vzdálenosti ve všech osách při každé iteraci. Taktéž existuje proměnná, která je s každou iterací inkrementována o jedničku a tedy slouží jako informace o počtu těchto hodnot. Na konci každého streamu je součet vzdáleností podělen počtem hodnot a výsledek této operace je zmiňovaná střední hodnota

$$\overline{x} = \frac{1}{n} \cdot \sum x_i \tag{1.19}$$

Rozptyl, jak už název napovídá, vyjadřuje do jaké míry jsou jednotlivé hodnoty náhodné veličiny rozptýleny okolo střední hodnoty. [24] V případě aplikace na odhad pózy se jedná o vyjádření toho, jak moc je obraz zašuměn a jak velké budou kmity výsledného odhadu.

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}} \tag{1.20}$$

RMSE je veličina, udávající rozdíl mezi predikovanou a reálnou hodnotou. [25] Je tedy standartní se s tímto parametrem setkat, při jakémkoliv článku, který se zabývá odhadem dat a tedy je programem uváděn i v tomto případě. Formula této proměnné je velice podobná rozptylu, ale je důležité chápat rozdíly mezi těmito dvěma statistikami. Zatímco rozptyl je hodnota, jak moc se jednotlivé členy náhodné veličiny liší od střední hodnoty, RMSE je hodnota, jak moc se hodnoty lší od predikce. Tedy zatímco u rozptylu je pracováno se střední hodnotou, jakožto jednoduchým bodem, kolem kterého je zjišťován rozdíl, u RMSE je tento střední bod sám náhodou veličinou. [26] Jednoduchou analogií k tomuto tématu by mohly být body, které prokládáme lineární funkcí - přímkou. Střední hodnota rozdíů na osel y jednotlivých bodů v daných hodnotách x se nazývá RMSE.

$$\sigma = \sqrt{\frac{\sum(y_i - \hat{y}_i)^2}{n}} \tag{1.21}$$

Posledním parametrem, kvůli kterému je celý tento proces spouštěn a který je třeba doplnit do lokalizačního programu je tzv. Kovarianční matice. Samotná kovariance vyjadřuje vzájemný směr pohybu dvou rozdílných hodnot. [27] Tedy pakliže aplikujeme definici na případ této diplomové práce, vyjadřuje kovariance, jakým způsobem se pohybují jednotlivé osy translace i rotace při nárůstu vzdálenosti a jakým způsobem jsou vzájemně ovlivňovány. Kovariance je vyjádřena jako:

$$Cov = \frac{\sum(x_i - \mu)^2}{n} = \sigma^2 \tag{1.22}$$

Pakliže je tedy vypočítána kovariance pro všechny kombinace páru os, které se mohou ovlivňovat a výsledky jsou vloženy do matice 6x6 (matice obsahuje jak kovarianci translace, tak i rotace), pak vznikne tzv. kovarianční matice, která je potřebná pro správné fungování lokalizačního programu.

$$\begin{bmatrix} Cov(x)^2 & Cov(xy) & Cov(xz) & Cov(xRx) & Cov(xRy) & Cov(xRz) \\ Cov(xy) & Cov(y)^2 & Cov(yz) & Cov(yRx) & Cov(yRy) & Cov(yRz) \\ ... & ... & Cov(z)^2 & ... & ... & ... \\ ... & ... & ... & Cov(Rx)^2 & ... & ... \\ ... & ... & ... & ... & Cov(Ry)^2 & ... \\ Cov(xRz) & Cov(yRz)^2 & Cov(zRz) & Cov(RzRx) & Cov(RzRy) & Cov(Rz)^2 \end{bmatrix} \tag{1.23}$$

1.7 Kalmanův filtr

Při vývoji v robotice nebo i jiných oborech se často potýkáme s různými druhy problémů při měření veličin. Jendím z hlavních takových problémů, který je předmětem mnoha diskusí a kvůli kterému je vyvíjeno velké úsilí pro jeho eliminaci, je šum. Dobrá filtrace většinou dělí kvalitní zařízení, software a obecně produkty od těch nekvalitních. Tato diplomová práce tomu není výjimkou. I zde je nutné se se šumem potýkat a jeho následky možná co nejvíce potlačit, za účelem získání nejpřesnějších možných výsledků. Naporsto přesnou lokalizaci nemůže být nikdy zaručena a to z mnoha různých důvodů. Řeč je například o obrazovém šumu, který znesnadňuje detekci, odlesky v obraze a podobné. Jedním způsobem, kterým je možné šum na datech minimalizovat, je použití takzvaného Kalmanova filtru. Slovní spojení "šum na datech"bylo použito záměrně. Správně nastavený Klamanův filtr je totiž schopen kompenzovat rozdíly mezi měřenou a reálnou hodnotou bez ohledu na to, jaká data jsou předmětem filtrace. K tomu je využíváno perfektní zanlosti systému a jeho reakce na podnět v určitém časovém úseku. Je ale nutné, aby šumy, které na systém působí měly nulovou korelaci a střední hodnotu. [29] [30] Jinak řečeno, se musí jednat o náhodný, bílý šum. Mimo to je také nutné zaručit, že systém je možné popsat lineárními rovnicemi, tedy se musí jednat o lineární systém nebo alespoň jeho co nejpřesněší aproximaci. Lineární systém je popsán následujícími rovnicemi. [28] [29] [30]

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ y_k &= Cx_k + z_k \end{aligned} \tag{1.24}$$

Ty popisují vyvoj stavů systému x a jeho výstupu y v čase. Matice A je takzvaná stavová matice a vyjadřuje změny jednotlivých stavů v momentech mezi x_k a x_{k+1}. B je nazývána vstupní matice a vyjadřuje, jakým způsobem a kam je připojen vstup systému. Poslední maticí je C. Té se přezdívá výstupní matice a popisuje, jakým způsobem jsou přivedeny jednotlivé stavy na výstup. [28] Z rovnic 1.26 je tedy možné jednoduše vyčíst, že nový stav x_{k+1} je odvozen od stavu v předešlém kroku x_k, který je vynásoben maticí jeho změn mezi jednotlivými kroky a také od vstupní

hodnoty systému. Stejně tak výstup systému je popsán stavem v nynějším kroku a způsobem, jakým jsou připojeny na výstup. Je možné si povšimnout, že z rovnic byly vynechány složky w_k a z_k. Tyto složky značí již zmiňovaný šum. Žádný reálný systém není dokonalý a se šumem je tedy nutno počítat jak na vstupu w_k, který je označován jako procesní šum, tak na výstupu z_k, kterému je zase přezdíváno šum měření. [28] [30] [31]

Stavy systému, které jsou vyjádřeny stavovým vektorem x_k, jsou často veličiny, jejichž hodnoty nelze přímo měřit. K tomu je využit výstup y, ze kterého jsou následně stavy dopočítány. Této hodnotě je ale možné důvěřovat pouze do určité míry kvůli všudypřítomnému šumu. [30] [31] Pakliže by bylo možné určitými statistickými metodamy šum minimalizovat na nepatrnou hodnotu, je možné získat z výstupu systému přesný odhad jeho stavů.Za tímto účelem musí být definovány rovnice Q, R, P jakožto součást návrhu filtru. Q reprezentuje kovarianční matici procesního šumu, R vyjadřuje kovarianční matici měřeného šumu a P zase kovarianční matici celkové chyby. [28] [29] [30] [31] Pro hlubší pochopení těchto matic je možné představit příklad reálného použití. Uvažujme měření pozici a rychlost letadla. Matice Q vyjadřuje v tomto případě nepřesnost reálné lokace a rychlosti z důvodů jako je vítr, turbulence a nebo momentálního výkonu obou motorů. Matice R značí nepřesnosti našeho měření. Příkladem můžou být nepřesná data z GPS navigace nebo jiných lokalizačních systémů. Poslední maticí je matice P. Ta vyjadřuje celkovou nejistotu esitmací pozice a rychlosti zkonstruovaného modelu vůči reálným hodnotám.

Jak již bylo řečeno, filtr je schopen se zbavit šumu pomocí znalosti systému a jeho výstupů v rámci určitého časového úseku. Tato věta v podstatě znamená, že filtr sleduje předešlé hodnoty výstupu a pomocí zmiňované znalosti systému je schopen upravit své parametry tak, aby byl schopen po určité době věrohodně predikovat hodnoty stavů v budoucích krocích. Práce Kalmanova filtru je tak rozdělena na dva kroky. Tou je predikce predikce. [28] [29] [30] [31]

$$\begin{aligned} \hat{X_k} &= A_{k-1}X_{k-1} + B_kU_k \\ \hat{P_k} &= A_{k-1}P_{k-1}A_{k-1}^T + Q_{k-1} \end{aligned} \tag{1.25}$$

a korekce.

$$\begin{aligned} V_k &= Y_k - H_k\hat{X_k} \\ R_k &= H_k\hat{P_k}K_k^T + R_k \\ K_k &= \hat{P_k}H_k^TR_k^{-1} \\ X_k &= \hat{X_k} + K_kV_k \\ P_k &= \hat{P_k} - K_kR_kK_k^T \end{aligned} \tag{1.26}$$

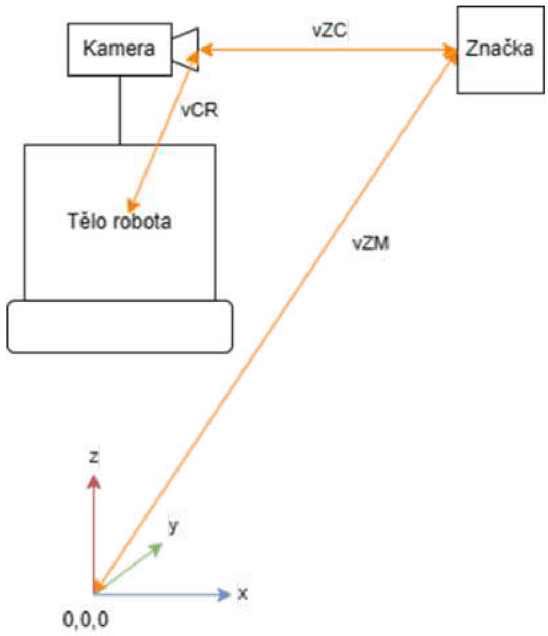
Krom známých matic P,Q a R a stavového vektoru X se zde vystihují také

- V_k - innovace stavů nebo také reziduum v kroku k
- K_k - Kalmánův zisk
- $\hat{X}_k$ - predikce stavového vektoru kroku k
- $\hat{P}_k$ - predikce matice P kroku k

Kalmánův filtr je široce používaným nástrojem s polem působnosti rozsahující od korekce polohy navigací při ztrátě signálu přes regulátory auto v automatizačním průmyslu až k jeho aplikaci ve vesmírném programu.

2 Výsledky studentské práce

Jak již bylo zmíněno, práce se zabývá lokalizací robota v prostoru, na základě optických značek. Doposud byla lokalizace řešena pomocí rychlosti otáčení kol daného robota. Tato metoda je velice jednoduchá, jelikož není potřeba vytvářet zvláště složité algoritmy k její implementaci. Problém ale nastává s přibývajícím časem. Po určité době se začnou výsledky lokalizace dle této metody lišit od skutečné pozice. Daný algoritmus nepočítá s možnými prokluzy kol a nejistotami měření. Zpětná oprava pozice již bohužel není možná. Výsledkem této práce by tak měl být algoritmus, který bude koexistovat s již implementovanou metodou a při zahlédnutí a rozpoznání specifikované značky jsou výsledky lokalizace korektovány. V teoretickém úvodu byl vysvětlen problém transformace a následná praktická část je založena převážně na tomto tématu. Představme si objekt v krychlové místnosti, jehož pozici je nutné zjistit. Jakožto první věc je nutné si určit počátek prostoru, ke kterému budou všechny pozice vztahovány. Dále v textu bude tato pozice nazývána jako „počátek“. Samotný prostor, ke kterému vztahujeme pozice robota a značek, a tudíž do kterého provádíme transformaci bude nazývána jako mapa (anglicky „map“). Když je tedy nyní určen počátek mapy, mohou být po místnosti rozmístěny značky. Polohu značek je nutné změřit a uložit do databáze, která je algoritmu přístupná. Kamera tyto značky zaznamená a následně je nutné provést již zmiňovanou transformaci.



Obr. 2.1: Transformace robota

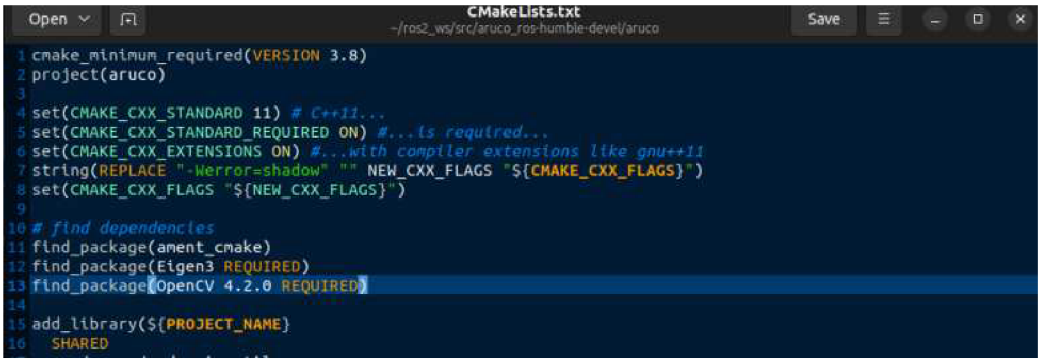
Obrázek výše znázorňuje, jak taková transformace vypadá. Tedy mezi počátkem a značkou se nachází předem určená, statická vazba vZM na základě umístění

značky. Značku zachytává kamera a její vzdálenost a natočení od značky je definováno vazbou vZC. Posledním větví transformace opět předem určená, statická vazba mezi kamerou a tělem robota – vCR. Pomocí takto definovaných vazeb je tzv. transformační strom spojitý a je možné provést transformaci těla robota do souřadnic mapy.

2.1 Využité knihovny

- ArucoRos
- tf
- UsbCam
- Image_pipeline
- OpenCV

Při instalaci knihoven nastal nejeden problém, které bylo nutné vyřešit před samotným programováním. Nicméně nejzásadnější problém nastal při instalaci knihovny ArucoRos. U té bylo zjištěno, že pro její správné fungování je vyžadována specifická verze knihovny OpenCV, kterou je OpenCV 4.2.0 . Tento požadavek je bohužel zmíněn pouze v instalačním souboru knihovny a je tak složité daný problém identifikovat.



Obr. 2.2: Ukázka instalačního souboru ArucoRos knihovny

2.1.1 Aruco ROS

Knihovna ArucoRos se stará o detekci a dekódování optických značek. Protože se jedná o stěžejní část práce, byla této knihovně věnována větší pozornost a je rozsáhle popsána v kapitole 1.5.1. Krom toho, že ArucoRos zpřístupňuje funkce Aruco v ROS frameworku, jsou navíc definovány proprietární formy zpráv, které knihovna využívá. Těmito zprávami jsou Marker message a Marker Array message. Marker message obsahuje informace o značce, která je vidět na obraze, pořízeného kamerou. Těmito informacemi jsou:

- Id
- Pozice
- Důvěryhodnost detekce

Tato zpráva dokáže ale nést informace pouze o jedné značce v obraze. Pakliže existuje možnost, že v záběru kamery budou v jeden moment pozorovány dvě a více značek zároveň, je nutné tuto zprávu zabalit do nové zprávy typu Marker Array. Marker Array (tak jak již název napovídá) obsahuje pole tyou Marker message a je tedy způsobem, jak přenášet informace vícero značek najednou.

2.1.2 tf2

Při práci na robotu představovala dříve transformace problém, která vyžadovala spoustu času a pozornosti pro naprosté odladění všech chyb. To vedlo ke tvorbě knihovny tf2, která je nyní standartním balíčkem při nainstalování ROS frameworku. Ta zbaví uživatele nutnosti ručního programování výpočtů a výrazně tak urychlí vývoj algoritmů. Při tvorbě praktické části této práce bylo nutné využít dva typy transformace - statickou a dynamickou. Statické transformace představuje například vazba kamery robota k tělu robota nebo vazba značky k počátku. Jsou to tedy transformace, jejichž vzájemná pozice se s časem nemění. Představitelem dynamické transformace může být pak vazba kamery robota ke značce. Tato vazba se mění v každém okamžiku v závislosti na pohybu robota a není tedy možné ji předem určit. K rozesílní zpráv o vazbách a pozicích jsou využívány broadcastové funkce TransformBroadcaster. Broadcast je typ posílání zpráv všemi směry, tedy zpráva o transformaci je rozeslána všem ostatním uzlům. Pokud ale chce uzel s transformací pracovat, musí být definován posluchač transformačních zpráv - TransformListener. Transformační zpráva posílána mezi broadcasterem a posluchačem je přesně definována a je typu `geometry_msgs/msg/TransformStamped`. Ta obsahuje:

- Hlavičku (obsahuje id referenčního systému (reference frame id))
- ID rámce potomka (child frame id)
- Transformace (typu `geometry_msgs/msg/Transform`)

Referenční rámec i rámec potomka představují dva různé souřadnicové systémy, mezi kterými je prováděna transformace. Řekněme, že existuje robot, jehož počátek leží na místě, kde byl poprvé spuštěn. Tento imaginární prostor představuje rámec potomka (child frame). Dále ale existuje místo v prostoru, které bylo určeno počátkem a tento bod v prostoru se neshoduje s bodem počátku v rámci potomka. Porostor v tuto chvíli představuje referenční rámec. Těchto systémů může najednou koexistovat vícero a informace o id rámce tedy určuje, mezi kterými prostory je tato transformace definována.

Samotná transformace je zprávou typu `geometry_msgs/msg/Transform` a obsahuje:

- Vektor translace
- Quaternion rotace

2.1.3 UsbCam

UsbCam je knihovna, která obstarává kamerový obraz a informace o kameře ostatním uzlům. Matice kamerového obrazu je posílána v topicu `/image_raw` a zmíněné informace v topicu `/camera_info`. Soubor informace o kameře vznikne pomocí kalibrace a obsahuje například následující informace:

```
video_device: "/dev/video0"
framerate: 30.0
frame_id: "camera"
pixel_format: "mjpeg2rgb"
image_width: 640...
```

2.1.4 Image_pipeline

Image_pipeline je balíček funkcí, jehož hlavním úkolem je zprostředkovat pomyslný tunel mezi surovými daty z kamery a aplikací počítačového vidění v rámci ROS. Mezi hlavní funkce, kvůli kterým je nutný balíček používat se řadí kalibrace kamery. Krom této funkce ale balíček obsahuje i jiné funkce zaměřené na práci s obrazem, jako je například zpracování stereo obrazu (využívání dvou kamer), odstraňování zkreslení obrazu, vypracovávání hloubkových nímků a další. Tyto funkce ale nebylo nutné v projektu použít.

2.2 Preparace a nastavení

Před začátkem samotného programování bylo nutné zaručit, že knihovny mají dostatečné množství informací pro požadovanou funkci. Tato kapitola se zabývá právě těmito přípravami. V textu výše již byla zmíněna kalibrace kamery. Mimo ní bylo ale nutné sestavit strom statických transformací a předpřipravit si prostředí, ve kterém se bude program (ROS package) nacházet.

2.2.1 Prostředí

ROS definuje přesnou formu prostředí, které je nutné dodržet pro správnou funkci programu. Výsledná struktura by měla vypadat následovně:

/ ..... kořenový adresář

```
└── Ros_workspace
    ├── build
    ├── install
    ├── log
    └── src
        ├── ArucoRos ..... příklad
        ├── UsbCam ..... příklad
        └── MyPackage ..... příklad
```

Prvním krokem je tvorba adresáře Ros_workspace a složky src. Následně je již ve složce src možné vytvořit nový balíček, pomocí:

Výpis 2.1: Příklad vytváření balíčku.

```
//C++:
ros2 pkg create --build-type ament_cmake --license Apache-2.0
<package_name>

//Python:
ros2 pkg create --build-type ament_python --license
Apache-2.0 <package_name>
```

Balíček je nutné posláze zkompilovat v adresáři Ros_workspace příkazem:

Výpis 2.2: Příklad kompilace.

```
colcon build --symlink-install
```

Je ale nutné nástroj colcon nainstalovat. Tento nástroj není součástí ROS. Pokud vše proběhlo bez chyb, zbytek adresářů bylo tímto krokem vytvořeno a struktura by měla mít podobu, jako to je ukázáno na začátku této kapitoly.

2.2.2 Kalibrace kamery

Pro kalibraci je nutné kameru prvně spustit pomocí:

Výpis 2.3: Příklad spouštění kamery pro kalibraci.

```
ros2 run usb_cam usb_cam_node_exe --ros-args
```

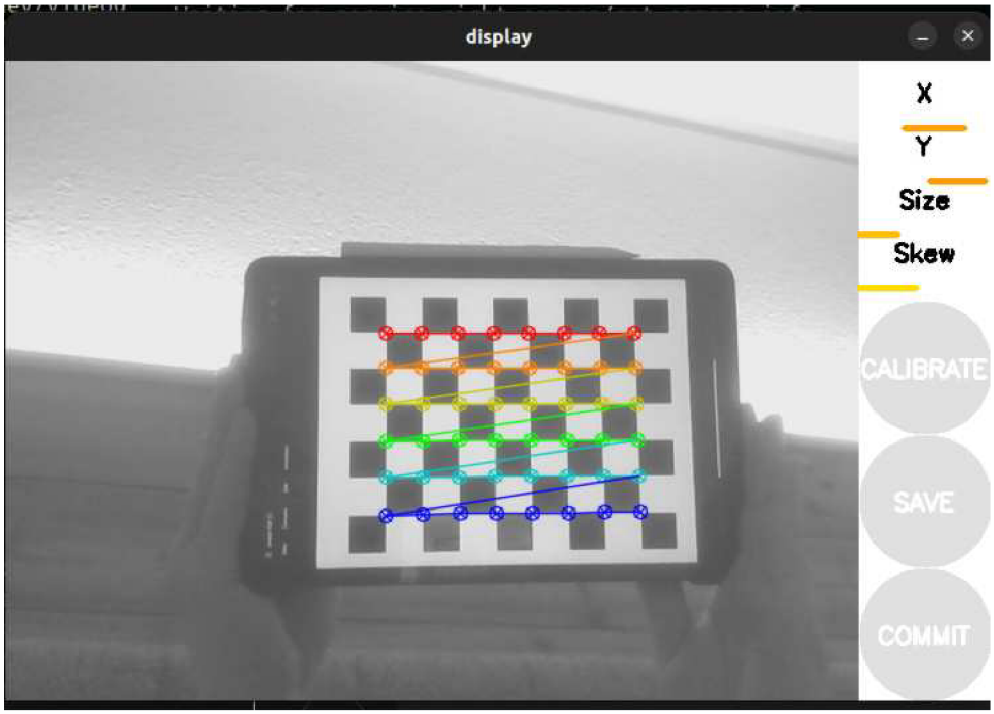
Nyní je možné pro kalibraci využít balíček camera_calibration z knihovny image pipeline.

Výpis 2.4: Příklad kalibrace kamery.

```
ros2 run camera_calibration cameracalibrator --size 8x6
--square 0.108 image:=''camera image topic''
camera:=''camera device''
```

Následně je otevřeno okno s obrazem kamery a třema zamčenýma tlačítkama na

pravé straně. V tento moment bylo nutné si stáhnout obrázek kalibrační šachovnice, vytisknout a položit před kameru. V příkazu bylo nutné sepcifikovat, jaký typ šachovnice je používán. Tedy je typu 8x6 políček a rozměr jednoho políčka je 1,08cm. Nutno podotknout, že pro správnou kalibraci byla šachovnice vytisknuta na papír a políčka změřena. Následující obrázek tak slouží pouze jako ukázka. Šachovnice je následně pozicována v různých vzdálenostech a částech obrazu tak, aby aplikace získala dostatek dat pro nastavení vnitřních parametrů kamery.



Obr. 2.3: Kalibrace kamery

Po nasbírání dostatku dat pro výslednou kalibraci, je na pravé straně zbarveno tlačítko "Calibrate" do zelena. Po zakliknutí jsou do zelena zbarvena i ostatní tlačítka. Nyní je nutné stisknout tlačítko "Save" a dané kalibrační soubory budou uloženy na disk. Při spouštění kamery musí být následně využíván spouštěcí příkaz s příznakem param file pro lokalizaci kalibračního souboru a jeho distribuci.

Výpis 2.5: Příklad spouštění kamery po kalibraci.

```
ros2 run usb_cam usb_cam_node_exe --ros-args
--param file:=''cesta k souboru''
```

2.2.3 Statické transformace

Statické transformace bylo nutné určit pro vazby kamera-tělo (robota) a počátek-značka. Veškeré tyto transformace byly uloženy do textového souboru, s mezerníkem,

jakožto dělícím znakem. Program tyto hodnoty načte a jejich transformace pošle všemi směry skrze broadcaster, jak to bylo popsáno v kapitole 2.1.2.

```
(object trans.x trans.y trans.z yee<rad> pitch<rad> yaw<rad>)
camera 0 1 -1 0 -1.57 -1.57
map 0 0 0 0 0 1
Id1 -1 3 1 -1.57 3.14 3.14
```

2.3 Průzkum knihovny Aruco_ROS

K průzkumu knihovny, její funkcí a celkové funkčnosti, bylo zapotřebí vytvořit dva programy. Prvním je tf_static_broadcaster, který se stará o rozeslání statických transformací, a tedy sestavením tzv. stromu transformací, které je poté možné zobrazit v programu Rviz2. V předchozí kapitole bylo řečeno, že statické transformace jsou uloženy v textovém souboru. Ty je dle podstaty nutné rozeslat alespoň jednou při sestavování stromu transformací. Programem jsou hodnoty načteny, rozeslány a následně není s hodnotami jakkoliv nadále pracováno. V tuto chvíli je možné topic ukončit, aby byl ušetřen výpočetní výkon.

Druhým, neméně zásadním programem, je tf_publisher. Tento program vznikl malými úpravami uzlu simple_single z knihovny aruco_ros. Úkolem simple_single není totiž zjišťovat vzdálenost kamery vůči počátku, ale vzdálenost značky od kamery, či počátku. Je také ale nutno podotknout, že popsané programy nejsou ve výstupu práce zahrnuty a jedná se pouze o vysvětlení, jak funguje detekce značek v kódu programů této knihovny. Podstatou tohoto uzlu je přijímat informace z obrazového vstupu kamery, rozeznávat v něm značky, počítat transformace a rozesílat je k dalšímu zpracování. Jelikož se jedná o tak stěžejní část práce, bude ji věnována větší pozornost, než tomu je u ostatních programů. Pro samotný výpočet je nutné, aby měl program dostatek informací. V tomto případě je nutné programu poskytnout minimálně kalibrační soubory specifické kamery. V prvním kroku je tato podmínka zkontrolována. Pakliže není soubor poskytnut, programem nebude vykonávána žádná činnost a bude tedy čekat, na jeho dodání, pomocí topicu "/camera_info". V opačném případě je zkontrolována podmínka druhá. Ta udává, že počet odběratelů výstupního topicu, obsahující výsledek lokalizace, nesmí být nulový. Opět zde platí, že pakliže tomu není vyhověno, program bude nečinně vyčkávat, než bude i tato podmínka splněna. Obě tyto podmínky v programu, jsou zahrnuty pro šetření výpočetního výkonu v případě, kdy je výpočet zbytečný. Pokud bude ale existovat alespoň jeden odběratel, je možné, aby byl výpočet vykonán. Obraz je

prvně překonvertován do RGB osmi bitového formátu a následně jsou v něm detekovány značky, pomocí příkazu ”detect” z knihovny Aruco ROS. Detekované objekty typu Aruco::marker, jsou poté uloženy do pole, nesoucí název "markers". Pokud toto pole není prázdné, tedy je jeho délka vetší než nula, je možné pokračovat. Od této chvíle jsou veškeré funkce prováděny pro každou značku zvlášť. Pokud je tedy cílem šetřit výpočetní výkon, je vhodné počet značek, které je možné najednou vidět v záběru kamery, omezit. Programem je nyní zkonstruována zpráva a je naplněna daty pro odeslání skrze broadcaster. To znamená naplnění informacemi, které jsou před samotným výpočtem dostupné. Těmi jsou rámec rodiče, rámec potomka a časová značka zprávy. Pokud se rámce rodiče a potomka neshodují, je mezi nimi vypočítána transformace, pomocí následujícího příkazu:

Výpis 2.6: Příklad spouštění kamery po kalibraci.

```
tf2::Transform transform =
aruco_ros::arucoMarker2Tf2(markers[i]);
this->marker_frame = "Id" + std::to_string(markers[i].id);

stampedTransform.header.frame_id = marker_frame;
stampedTransform.header.stamp = curr_stamp;
stampedTransform.child_frame_id = camera_frame;
tf2::toMsg(transform, stampedTransform.transform);
tf_broadcaster_->sendTransform(stampedTransform);
...
getTransform(reference_frame, camera_frame,
transform_stamped);
...
tf2::toMsg(transform_stamped.transform, poseMsg.pose);
pose_pub->publish(poseMsg);
```

V tomto úryvku kódu se nachází hned několik důležitých příkazů, které popisují celou detekci. Tím prvním je naplnění proměnné transform. Hodnota "markers[i]"z úryvku sice obsahuje translašní vektor i rotační quaternion. Do daného translačního vektoru ale není rotace zakomponována a jedná se tedy o dva samostatné celky, které musí být určitým způsobem spojeny. Více k této problematice, bude rozebráno dále v textu, v kapirole 2.5.4. Jakkmile je známa tato hodnota, je vytvořena zpráva pózy, je naplněna důležitými informacemi a je odeslána do stromu transformací. Poté je zavolána funkce getTransform(), která vrací pózu kamery vůči počátku a i ta je následně vysílána, nyní ale pouze pro kontrolu, jelikož ve stromu již existuje vazba kamery vůči značce.

Pro samotným výpočet transformace mezi tělem robota a počátkem, byl vytvořen

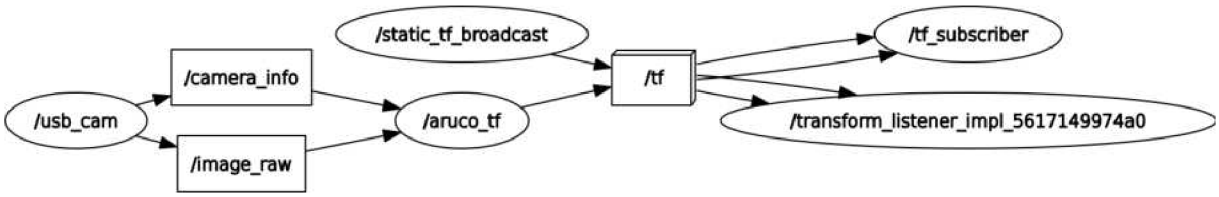
třetí program a tím je tf_subscriber. Jelikož je programu nyní známa pozice kamery v souřadnicovém systému mapy, je výsledný výpočet transformace jednoduchou úlohou, jelikož mezi tělem robota a kamerou je definována statická transformace. Tento program vznikl pro ověření funkce a správnosti výsledku celkového experimentu, popsaného v této kapitole. Bylo by samozřejmě možné transformaci těla vypočíst již v tf_publisheru, nicméně jelikož se jedná o experiment, johž cílem bylo se s knihovnou seznámit, přišlo mi žádoucí, vyzkoušet předávání transformačních hodnot mezi uzly. S výsledkem již není nutné dále nijak pracovat, a tak je pouze vypisován do konzole. Programem je využívána funkce "lookup_transform"k dohledání výsledku.

Výpis 2.7: Příklad vypisování výsledné transformace.

```
t = self.tf_buffer.lookup_transform(self.reference_frame,
self.child_frame, rclpy.time.Time())

self.get_logger().info(f"Translation:
{t.transform.translation}, Rotation: {t.transform.rotation}")
```

Reference frame je v tomto případě bod počátku a child frame představuje tělo robota.

Výsledný program má následně strukturu, která je vidět na obrázku níže. K vykreslení grafu byl použit nástroj RQT graph (jež je součástí většího balíčku funkcí RQT), který je součástí prostředí ROS a slouží hlavně k vizualizaci komunikujících uzlů, a tedy zjednodušení odstraňování problémů v programu.



Obr. 2.4: Graf aplikace

Ikyž se i tato vizualizace může zdát zprvu nepřehledná a nicneříkající, rqt_graph je velice mocným nástrojem, s velice pozitivními přínosy k práci. Jelkož většina popsaných programů spoléhají na přijetí zprávy k tomu, aby mohly fungovat, je při chybě občas složité hledat, která z komunikací neproběhla správně. V grafu je dále možné vidět, jak celý program funguje, což zase může pomoct progamátorům, navazující na cizí práci. Programy ROS mohou obsahovat desítky spolu komunikujících uzlů a nabývat tak obrovských rozměrů. Právě tehdy jsou opravdu odhaleny výhody tohoto nástroje. V tomto případě, existují pouze čtyři komunikující programy. Usb_cam poskytuje hlavnímu uzlu informace o kameře a surová data obrazu. Ty jsou

zpracovány a společně se statickými transformaci z programu static_tf_broadcast, posílány skrze topic /tf do uzlu tf_subscriber. Mimo tohoto příjemce, existuje ale v grafu ještě jeden posluchač. Tím je již zmiňovaný program k vizualizaci pozic rámců v prostoru - Rviz2.

2.3.1 Soubor CmakeLists

Psaní kódu je pouze částí tvorby programu. Dnes již často opomíjeným, ale nijak méně důležitým úkonem je samotná kompilace. Pro psaní kódu je větsinou zvolen určitý typ programovacího jazyka vyšší úrovně (higher level programming language), který je dobře čitelný pro člověka. Počítač tomuto jazyku ale není schopen porozumět a je tedy nutné kód přeložit do podoby binárního souboru, který již je možné procesorem zpracovat. Tento překlad se nazývá kompilace. CmakeLists je soubor, který popisuje, jakým způsobem se má daný projekt zkompilovat. Mimo jiné CmakeLists obsahuje informace o tom, jaké knihovny je potřeba nainstalovat, kde se soubor s kódem nachází nebo jaký kompilátor (verze) je zvolen. Pro spuštění programu je tak nutné, aby byl zahrnut v CmakeLists, jinak jeho kompilace nebude provedena a program tak nebude spustitelný. Do CmakeListu aruco_ros bylo potřeba přidat program tf_publisher, který byl v rámci experimentu vytvořen. Samotné přidání programu, pro jeho správnou kompilaci má v tomto případě následující strukturu.

Výpis 2.8: Příklad CmakeLists kódu

```
add_executable(myNode src/MyNode.cpp)

target_include_directories(myNode PUBLIC include)

target_include_directories(myNode SYSTEM PUBLIC
${OpenCV_INCLUDE_DIRS})
```

2.3.2 Launch soubor

V programu je možné definovat proměnnou, která je například nutná měnit dle specifického zařízení, na kterém je spuštěna. Představme si, že v programu existuje proměnná, určující rozlišení vstupního obrazu. Je samozřejmě možné tuto informaci zjistit, proměnnou v kódu najít a následně přepsat její hodnotu, nicméně takový program je pak specificky nastaven pro jedno zařízení a při změně je nutné, tento postup opakovat. Aby bylo tedy možné uzel spustit s jinými parametry, je definována proměnná jako:

Výpis 2.9: Příklad definice proměnné

```
this->declare_parameter<int>("rozliseni_X", 250);
```

Tento příkaz znamená, že hodnota parametru rozliseni_X bude deklarována mimo kód programu. Pokud se tomu tak nestane, je proměnné přiřazena výchozí hodnota 250. Deklarace lze dosáhnout dvěma způsoby. Prvním je spuštění uzlu v příkazovém řádku konzole s příznakem proměnné a jeho hodnoty. Tento způsob je sice jednodušší, ale pokud program obsahuje desítky hodnot, je nutné při každém spuštění všechny hodnoty určovat znovu, což může být poměrně zdlouhavé.

Výpis 2.10: Příklad definice pomocí příkazového řádku

```
ros2 run myNode --ros-args --rozliseni_X := 480
```

Druhým způsobem je vytvoření spouštěcího souboru (Launch file) v jazyku Python. Launch file je samostatný program, jehož úkolem je spouštění daného program a to včetně deklarace proměnných, které jsou určeny jako proměnlivé tak, jak to bylo popsáno výše v textu.

Výpis 2.11: Příklad definice pomocí launch file

```
rozliseni_X_arg = DeclareLaunchArgument(
'rozliseni_X', default_value=480
)
...
ld = LaunchDescription()
ld.add_action(rozliseni_X_arg)
return ld
```

Tímto způsobem je také možné dynamicky měnit topicy, které jsou v kódu odebírány. Není tak nutné, aby byl topic, kterým je poskytován například kamerový obraz, pojmenován "image_raw". To je důležité, pakliže je uzel vkládán do již existující struktury na určitém zařízení. Pro implementaci je totiž možné změnit pouze tyto názvy ve spoštěcím souboru a není tak nutné, aby bylo jakýmkoliv způsobem zasahovíno do kódu programu.

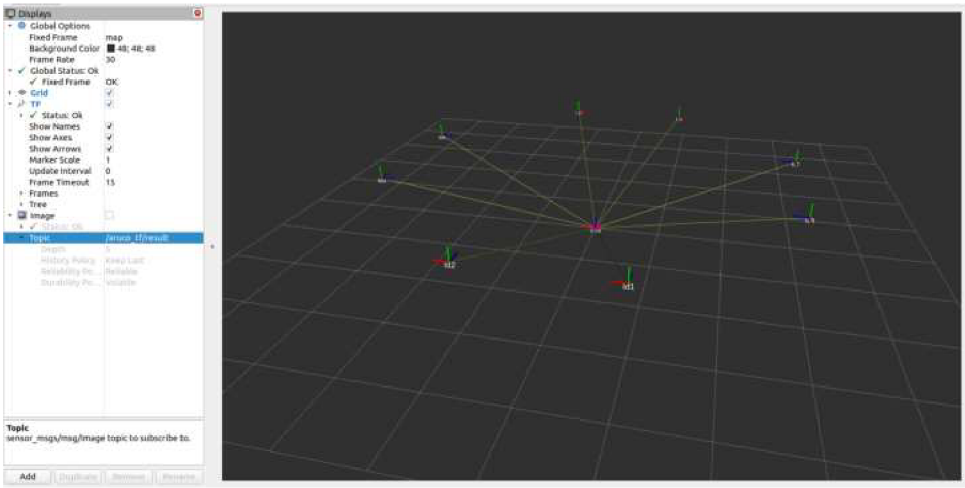
2.4 Simulace

Prvním krokem při simulaci je spuštění tf_subscriberu. Ten musí být aktivní již při sestavování stromu transformací a je tedy vhodné, aby byl spuštěn jako první. Při simulaci je pracováno v prostředí programu Rviz2, což je vizualizační nástroj, vytvořený pro toto použití v prostředí ROS. Rviz2 je spouštěn pomocí:

Výpis 2.12: Příklad spouštění simulačního programu.

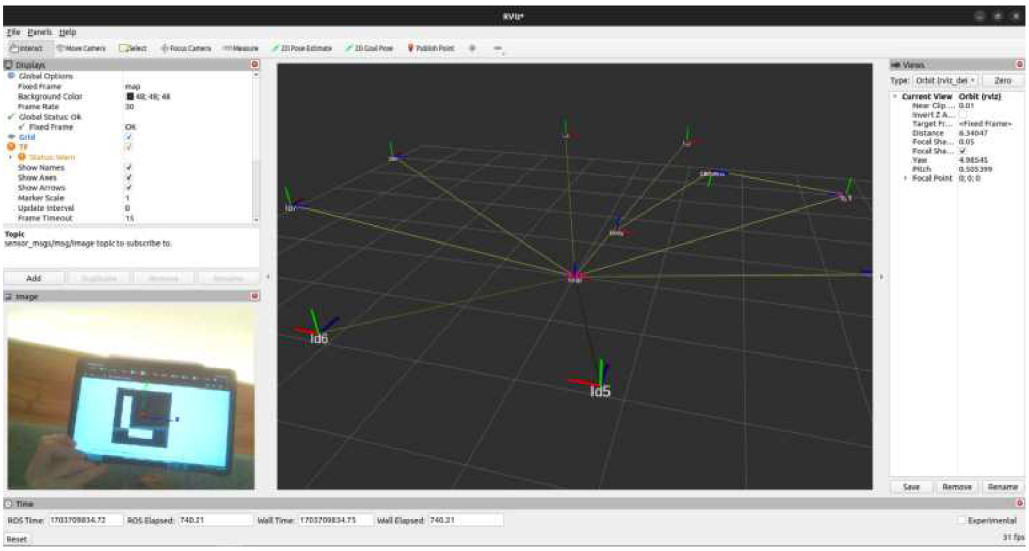
```
ros2 run rviz2 rviz2
```

Po zadání příkazu do konzole je otevřeno hlavní okno. V levé části programu, je v menu možné si zvolit topicy, které mají být programem odebírány a vizalizovány. Mimo transformací, je tak možné zobrazit kamerový vstup, 3d objekty a jiné. V prostředním okně je zobrazen trojrozměrný prostor, který je prozatím prázdný, jelikož doposud nebyly distribuovány žádné transformace. Je tak nutné, aby byl spuštěn program static_tf_broadcaster. Pokud je správně nastaven soubor, obsahující seznam póz objektů, měl by být v programu automaticky sestaven, a tedy i být viditelný strom transformací. Transformace byly definovány tak, aby simulovaly tvar reálné místnosti. To znamená, že všechny značky jsou pozicovány do tvaru čtverce o rozměru 6x6 metrů a jsou natočeny směrem k počátku ve výšce jednoho metru nad zemí.



Obr. 2.5: Simulace transformačního stromu

Nyní je možné, aby byl spuštěn hlavní uzel - tf_broadcaster. Rviz2 již automaticky naslouchá transformacím. Mimo ně je ale nutné odebírat také kamerový výstup tf_publisher uzlu. Toho je docíleno, pomocí menu v levé části obrazu, tlačítkem "Add". Po patřičném nastavení, byla před kameru postavena značka. Její hodnota může nabývat kteréhokoliv čísla z dostupných osmi, jež tvoří simulovanou místnost. Pakliže je značka rozpoznána, k čemuž je potřeba, aby byla dobře viditelná, nepřekrývaná kterýmkoliv objektem a umístěna na rovném podkladu, v programu je zobrazen model robota s vazbou k příslušné značce. Model robota je tvořen dvěma body (kamera a tělo), mezi kterými je definována statická vazba. Kamera je oproti tělu robota posunuta jeden metr v ose +Z a jeden metr v ose +X.



Obr. 2.6: Výsledná simulace

Značka v obraze byla různě pozicována a byl sledován program s modelem robota. Transformace a odhad vzdálenosti kamery od značky se prokázalo býti poměrně přesné. Jediným problémem, je okolí necitlivosti detekce natočení kolem nulového bodu. Je prakticky nemožné detekovat nulové natočení značky a výsledná transformace je tak vždy mírně vychýlena směrem do kladných, či záporných hodnot osy Z.

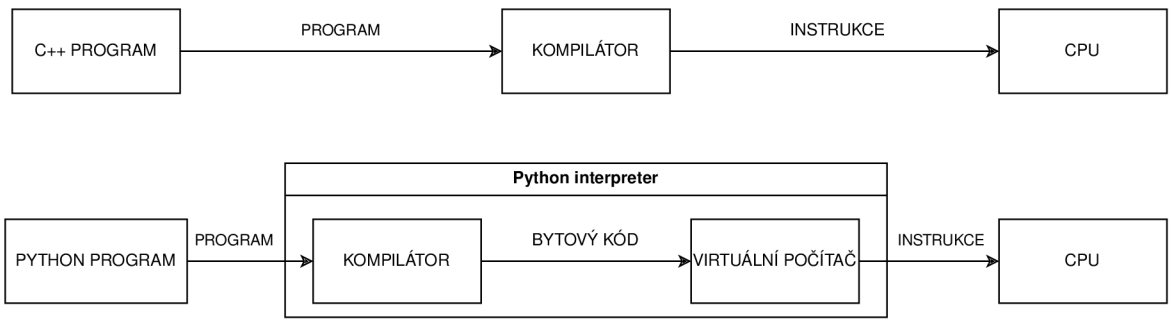
2.5 Lokalizační balíček Aruco ROS locator

Po patřičném vypracovaní simulační fáze, kdy byl koncept detekce dostatečně pochopen, bylo možné započít vývoj lokalizačního balíčku, který je výstupem této práce. Jméno tohoto balíčku bylo zvoleno tak, aby dostatečně reflektovalo knihovnu, na kterou bylo navazováno (aruco_ros) a zároveň bylo jasné, co je jeho cílem. Již představený uzel simple_single, využitý k simulaci, byl nahrazen programem ze stejné knihovny, který nese název marker_detector. Ten plní podobnou funkci jako simple_single, nicméně je možné detekovat z obrazového výstupu kamery vícero značek najednou. Výstupem z marker_detectoru jsou již představované zprávy knihovny aruco_ros - Marker a MarkerArray. Před začátkem vytváření vlastního programu bylo ale nutné vyřešit důležitou otázku. Onou otázkou bylo, jaký jazyk zvolit pro vývoj konečného projektu.

2.5.1 Výběr jazyka

Jak již bylo zmíněno v kapitole 2.3, ROS balíček lze vytvářet pomocí dvou rozdílných jazyků. Těmi jsou C++ a Python. Oba jazyky jsou objektově orientované, neboli zkráceně řečeno, využívají OOP (objektově orientované programování). Jádro OOP jazyků je tvořeno, jak už samotný název napovídá, objekty. V programu jsou funkce, proměnné a jiná data uchovávány uvnitř tzv. tříd. Z těchto tříd mohou být následně vytvářeny objekty v kódech jiných programů, obsahující své vlastní třídy a tímto způsobem zdědit jejich funkce a jiná data. V tomto ohledu jsou jazyky stejné, a tedy výběr kteréhokoliv z nich, nepředstavuje žádnou markantní výhodu.

Velkou výhodou C++ je jeho bezprostřední rychlost. [34] [35] Bez ohledu na volbu jazyka, je nutné po napsání kódu, program zkompilovat. Kompilaci je možné si představit jako překlad příkazů z formy, které může porozumět člověk do strojového kódu, jemuž pro změnu rozumí CPU. Tanto funkce je zprostředkovávána tzv. Kompilátorem. Následně je vygenerován spustitelný .exe soubor, jehož obsahem je právě zmíněný zdrojový kód. Zkráceně řečeno je program po spuštění vykonáván přímo na CPU. U Pythonu je situace značně odlišná. I zde je kompilátor významnou součástí při spouštění programu. Místo zdrojového kódu jsou ale instrukce překládány do tzv. bytového kódu. CPU není ale schopno tomuto kódu porozumět, a tak je k jeho spuštění vyždavoána tzv. Python Virtual Machine (PVM). Kompilátor i PVM jsou podmnožininami celku, které je přezdíváno interpreter. PVM je určitý typ virtuálního operačního systému, který následně vykonává instrukce z bytového kódu na hadrwaru zařízení. To znamená, že bytový kód je v PVM přeložen do konečného strojového kódu. U Pythonu je důležité, že tento proces je prováděn řádek po řádku, při běhu programu, díky čemuž jazyk získává své charakteristické charakteristické vlástnosti, za cenu nižší rychlosti. [37] [38]



Obr. 2.7: Kompilace C++ vs Python

Dalším rozdílem, který je mezi těmito jazyky k nalezení, se týká práce s pamětí.

C++ nabízí programátorovi mnohem bližší interakci. [36] Vlastností tohoto přístupu ale je, že u C++ není využit garbage collector. Garbage collector je široce používaný algoritmus, jehož podstatou je kontrola proměnných, jiných data a kódu, který k těmto datům přistupuje (čte, zapisuje, či jinak pracuje). [38] Pro práci s těmito proměnnými, je nutné, aby byly načíteny do paměti. Pokud s nimi ale program již žádným způsobem neinteraguje, není důvod aby byly v paměti dále uchovávány a v tomto případě jsou data odstraněny garbage collectorem. U Pythonu je na druhou stranu garbage collector využíván, za cenu nižší kontroly nad pamětí programu.

Po zvážení všech kladů a záporů, bylo rozhodnuto, že výsledná knihovna této diplomové práce bude vypracována v jazyce Python. Samozřejmě přívětivost jazyka, který se svou velkou komunitou, nabízí nepřeberné množství knihoven, byl jedním z faktorů, který měl při výběru velkou váhu. Mimo tento důvod bylo ale naznáno, že dané úkony, které jsou lokalizačním programem prováděny, nejsou až natolik výpočetně náročné, aby hrála menší rychlost Pythonu natolik významnou roli. Pakliže by bylo nutné, programem obstarávat samotnou detekci, bylo by použití C++ vhodné. S vědomím ale, že o detekce je zprostředkovávána knihovnou aruco_ros, byl v tomto ohledu upřednostněn Python. Posledním parametrem, kterým bylo rozhodnuto ve prospěch jazyka, je použití garbage collectoru. Jak již bylo řečeno, s C++ je programátorovi nabízena mnohem užší kontrola nad pamětí programu: Nicméně příčinou je také vysoký nárust obtížnosti s její správou. Malé chyby v kódu mohou zapříčinit katastrofické následky, které se mohou projevit s narůstajícím časem. Jistota, že paměť je spravována garbage collectorem byla tedy brána spíše jako výhoda. Tento argument ještě nabírá na váze s omezenou pamětí robotů, s kterými je často v oboru robotiky pracováno.

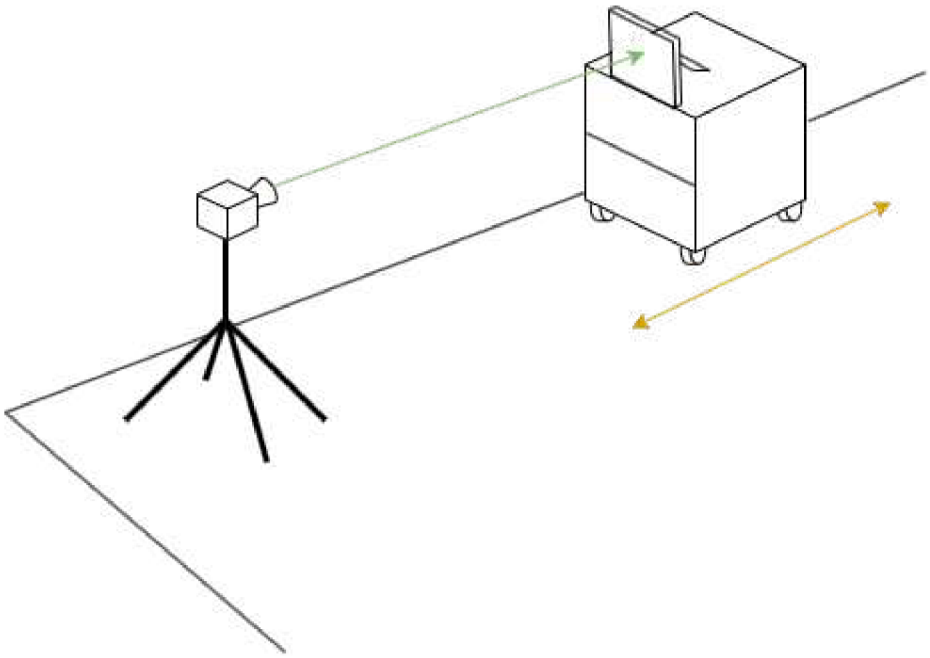
2.5.2 Měření přesnosti

V rámci diplomové práce je samozřejmě důležitým tématem odchylka. Konečná póza, která je detektorem a nebo i vytvořeným programem poskytována, nemůže být nikdy zcela přesná. Hlavním důvodem, kterým je k tomuto bezmála nepříjemnému jevu přispíváno, je tzv. obrazový šum. Ten je zapříčiněn náhodně pohybujícími se elektrony na snímači. Těmito elektrony jsou následně ovlivňovány i sousední částice, neboli obrazové body, a vzniká tak zkreslení obrazu. Na digitální fotografii je pak možné si šumu povšimnout pouhým okem a má charakter náhodné, barevné mlhy. Tato "mlha"je často mnohem intenzivnější na obrazu z levnějších, méně kvalitních kamer. Ty totiž krom horšího senzoru, disponují často také méně kvalitním filtrem, který býva používán právě na odstranění onoho šumu. I bez tohoto šumu by však nebyla lokalizace naprosto přesná. Když je pominut fakt, že naprosté přesnosti

není možné z fyzikálních důvodů dosáhnout, kvůli nepřesnosti měřících přístrojů, je dalším představitelem problém samotná projekce 2d obrazu do 3d a naopak. I kdybychom byli schopni kameru, pomocí šachovnice, zkalibrovat na každý milimetr dohledového prostoru kamery, stále se jedná pouze o projekci. To dává vzniku celá řady zaokrouhlovacích chyb, které nyní rozdělím na dva typy. Prvním typem je početní zaokrouhlovací chyba. Ta vzniká při výpočtu, kdy jsou programem zaokrouhlována určitým způsobem desetinná místa, které tvoří výslednou chybu. Druhým typem jsou pak zaokrouhlovací chyby obrazové. Každá projekce je limitována samotným rozlišením kamery. Jelikož je výsledkem kalibrace, namapování každého pixelu obrazu na 3d prostor, je jasné, že objekt nemůže ležet ve 2/3 pixelu, ale je jeho pozice zaokrouhlena na pixel celý. Lokalizace tedy nikdy nemůže nabývat hodnot nulové odchylky, nicméně je nutné chybu minimalizovat na hodnotu, která je přijatelná v rámci aplikace daného projektu (v tomto případě této diplomové práce).

Z tohoto důvodu je lokalizační aplikací posílána zpráva PoseWithCovariance, jež obsahuje výslednou pózu společně s kovarianční maticí. Tou jsou udávány informaci o přesnosti detekce a bylo nutné ji zkonstruovat, pomocí reálného experimentu. Měření je ale často drahý a neméně zdlouhavý proces. Tématem této diplomové práce není měřením přesnosti detekce, nýbrž jeho reálná aplikace. Byl tudíž vybrán článek [43], ve kterém jsou obsažena všechna potřebná data. Bylo nicméně stále nutné ověřit jejich věrohodnost, a tak byl zkonstruován jednoduchý pokus. Pokus byl sestaven z kamery ve statickém bodě, upevněnou na stojanu a posuvné části nábytku, na který byla značka vodorovně postavena na tvrdý podklad a důkladně zaaretována. Nákres výsledného pokusu je vidět na obrázku níže.

Tento experiment rozhodně nemá kvality pro uvedení ve kterékoliv vědecké práci, jako závěr o přesnosti. Jako kontrola hodnot z uvedeného článku, pomocí reálných dat z měření, byl ale tento pokus postačující. Nebylo totiž možné zajistit nulovou odchylku (a tudíž i přesnost) v žádné z os rotace a dvou os translace. Jediná hodnota, která tedy mohla být měřena s určitou přesností, je vzdálenost v ose Z. Samozřejmě nebylo možné přesně zrekreovat experiment z [43], jelikož byla použita odlišná instrumentace. Data měly ale řádově stejné hodnoty a tudíž se dá předpokládat, že experiment lze brát jako výchozí. Experiment následně probíhal posuvem stolku podél stěny, do předem určených vzdáleností a využitím nástroje ros_bag, díky němuž jsou data nahrávána a mohou být použita pro měření statistických veličin. Každá vzdálenost byla následně kontrolována pomocí laserového měřiče vzdálenosti. K výpočtu byl použit skript, který je součástí výsledné práce.



Obr. 2.8: Pokus měření přesnosti

Skript pro měření statistických veličin

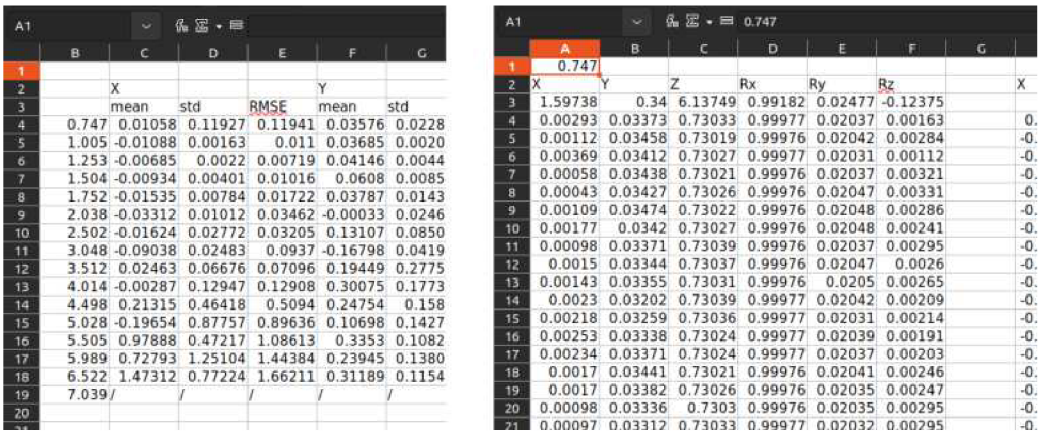
Tento program byl koncipován tak, aby měl uživatel možná co nejmenší práci a tudíž i největší komfort, při určování statistických veličin odhadu lokalizace. Prvním krokem je specifikovat v launch filu 7 hlavních komponentů. Jak již bylo řečeno, pro program byly nahrány obrazové streamy skrze nástroj ros_bag. Pro následující uživatele je tedy postup stejný, a tedy je prvně nutné vytvořit ros_bag soubory pro každou měřenou vzdálenost. Následně je potřeba tyto soubory umístit do složky a cestu k ní specifikovat v launch file. Složka posléze musí odpovídat následující struktuře.

```
/ ..... kořenový adresář
└── mereni
    ├── 0.757 ..... zde doplňte vlastní měřené vzdálenosti
    │   └── rosbag. ..... soubor ros_bag
    ├── 1.256
    │   └── rosbag. ..... soubor ros_bag
    ├── 1.516
    │   └── rosbag. ..... soubor ros_bag
    └── ...
```

Jednotlivé složky je nutné pojmenovat tak, aby jejich názvy odpovídaly vzdálenostem měření v metrech. Standartně tyto vzdálenosti odpovídají translaci v ose Z, je ale možné v launch file specifikovat osu, která je tímto zamýšlena pomocí proměnné "folders coresponds to axis". V launch file jsou nadále specifikovány reálné

hodnoty ve všech osách translance i rotace, které uživatel doplní, dle svého měření. Nehledě na tom, která osa je specifikována ve "folders_corresponds_to", je možné nechat pole specifikace reálných hodnot prázdné. V případě osy, která odpovídá "folders_corresponds_to", jsou jednotlivé vzdálenosti obdrženy z názvů složek a tuto hdonotu tak uživatel není nucen vyplňovat. V případě jakékoliv jiné osy, je předpokládáno, že se jedná o osu která je zamýšlena s nulovou odchylkou. Její hodnoty jsou tak následně nastaveny na 0. Pakliže by byla uživatelem naplněna složka ros_bag soubory tak, jak je specifikováno v textu a hodnoty v launch file by byly ponechány výchozí, program bude předpokládat, že bylo pohybováno pouze s osou Z a ostatní osy mají nulový posun i rotaci.

Ke spuštění programu pro měření statistických veličin je využíván marker_detector z balíčku aruco_ros a lokalizátor z balíčku této diplomové práce. Pro uživatele je ale důležité změnit hodnotu 'reference_frame' pole v launch file lokalizátoru a nastavit jméno rámce značky, která je využívána k měření. Poslední nutný úkon před spuštěním tohoto uzlu, je naplnění stromu statických transformací. Tento krok je nutný, jelikož je oběma programy využíván tf_buffer, kterým je ve své paměti strom ukládán a následně je s touto pamětí operováno v rámci programu. Pakliže jsou všechny tyto kroky splněny, je možné spustit uzel měření statistických veličin. Tím je jako první rozeslána zpráva s kalibračním souborem kamery, jehož umístění bylo specifikováno v launch file. Tento krok byl přidán, kvůli nekonzistenci přijetí kalibrační zprávy z ros_bag souboru. Pokud byla zpráva přijata uzlem marker_detctor, je automaticky načten první ros_bag soubor a uživateli je oznamováno, že byla přijata transformace. Po dokončení prvního streamu, je ihned načten druhý a tento cyklus je opakován, dokud není programem doiterováno skze všechny dostupné soubory. Výsledkem jsou dva grafy typu .png a jeden sešit typu excel. V tomto Excel souboru jsou uloženy jak jednotlivé změřené hodnoty pro případ, kdy by uživatel vyžadoval výpočet veličin nad rámec těch, které jsou poskytovány, tak statistické veličiny počítané uzlem.



	B	C	D	E	F	G
1						
2		X			Y	
3		mean	std	RMSE	mean	std
4	0.747	0.01058	0.11927	0.11941	0.03576	0.0228
5	1.005	-0.01088	0.00163	0.011	0.03685	0.0020
6	1.253	-0.00685	0.0022	0.00719	0.04146	0.0044
7	1.504	-0.00934	0.00401	0.01016	0.0608	0.0085
8	1.752	-0.01535	0.00784	0.01722	0.03787	0.0143
9	2.038	-0.03312	0.01012	0.03462	-0.00033	0.0246
10	2.502	-0.01624	0.02772	0.03205	0.13107	0.0850
11	3.048	-0.09038	0.02483	0.0937	-0.16798	0.0419
12	3.512	0.02463	0.06676	0.07096	0.19449	0.2775
13	4.014	-0.00287	0.12947	0.12908	0.30075	0.1773
14	4.498	0.21315	0.46418	0.5094	0.24754	0.158
15	5.028	-0.19654	0.87757	0.89636	0.10698	0.1427
16	5.505	0.97888	0.47217	1.08613	0.3353	0.1082
17	5.989	0.72793	1.25104	1.44384	0.23945	0.1380
18	6.522	1.47312	0.77224	1.66211	0.31189	0.1154
19	7.039	/	/	/	/	/
20						

	A	B	C	D	E	F	G
1	0.747						
2	X	Y	Z	Rx	Ry	Rz	X
3	1.59738	0.34	6.13749	0.99182	0.02477	-0.12375	
4	0.00293	0.03373	0.73033	0.99977	0.02037	0.00163	0.
5	0.00112	0.03458	0.73019	0.99976	0.02042	0.00284	-0.
6	0.00369	0.03412	0.73027	0.99977	0.02031	0.00112	-0.
7	0.00058	0.03438	0.73021	0.99976	0.02037	0.00321	-0.
8	0.00043	0.03427	0.73026	0.99976	0.02047	0.00331	-0.
9	0.00109	0.03474	0.73022	0.99976	0.02048	0.00286	-0.
10	0.00177	0.0342	0.73027	0.99976	0.02048	0.00241	-0.
11	0.00098	0.03371	0.73039	0.99976	0.02037	0.00295	-0.
12	0.0015	0.03344	0.73037	0.99976	0.02047	0.0026	-0.
13	0.00143	0.03355	0.73031	0.99976	0.0205	0.00265	-0.
14	0.0023	0.03202	0.73039	0.99977	0.02042	0.00209	-0.
15	0.00218	0.03259	0.73036	0.99977	0.02031	0.00214	-0.
16	0.00253	0.03338	0.73024	0.99977	0.02039	0.00191	-0.
17	0.00234	0.03371	0.73024	0.99977	0.02037	0.00203	-0.
18	0.0017	0.03441	0.73021	0.99976	0.02041	0.00246	-0.
19	0.0017	0.03382	0.73026	0.99976	0.02035	0.00247	-0.
20	0.00098	0.03336	0.7303	0.99976	0.02035	0.00295	-0.
21	0.00097	0.03312	0.73033	0.99977	0.02032	0.00295	-0.

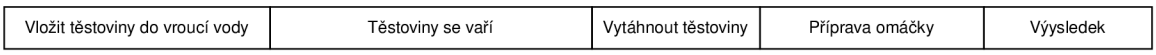
Obr. 2.9: Výsledný Excel tabulka a hodnoty

2.5.3 Asynchronní programování

Výsledným programem jsou využíványi funkce vícevláknového, asynchronního programování. Před samotným popisem kódu, je tedy nutné vysvětlit, co tento styl programování obnáší a jaké jsou jeho výhody a nevýhody. V kapitole 2.5.1 již bylo zmíněno, že kód, psaný uživatelem, je pomocí kompilátoru přeložen do instrukcí, kterým je CPU schopno porozumět a vykonat je. Je tedy logické, že jsou tyto příkazy obsluhovány sekvenčně za sebou tak, jak jsou definovány v programu. Tento fakt ale není v případě asynchronního programování uplatněn. Každé CPU obsahuje své vlastní jádro. Toto jádro dovoluje procesorům zpracovávat úkoly, zadané uživatelem a jiné systémové požadavky. Dříve procesory obsahovaly pouze jedno jádro. Tedy procesor jako takový mohl vykonávat pouze jednu instrukci v daný moment. Dnes jsou ale standartně dostupné CPU s dvěmi, čtyřmi i vícery jader. [40] Skrze asynchronní programování je tedy programátor schopen utilizovat tento vyšší počet jader a optimalizovat tak běh programu výmennou, za vyšší náročnost při programování i vykonávání na procesoru. Dané instrukce tak nemusí být vykonávány postupně tak, jak jsou definovány v programu. [39] Zdlouhavé procesy mohou být vykonávány na jednom jádře, zatímco ty jednodušší jsou rychle prováděny na jádrech jiných.

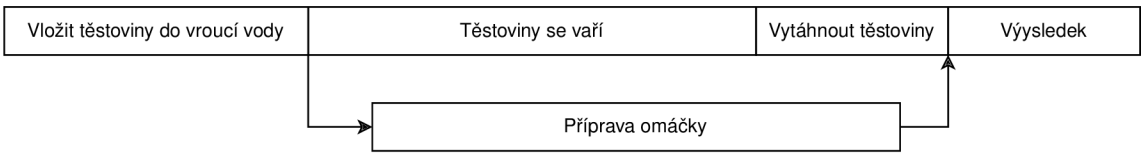
Asynchronní programovaní je možné si lépe představit na příkladu z reálného života. Většina činností, kterou člověk na denní bázi provádí, jsou automaticky vykoávány assynchronně, aniž by daný jedinec musel chápat význam tohoto slova. Pro příklad je možné uvést vaření těstovin. Představme si kuchyň, kde se nachází sporák s linkou. Sporák má sice čtyři plotýnky (představující jádro procesoru), ale v tomto případě bude využívána pouze jedna. Na onom sporáku je již v hrnci připravena vroucí voda. Pokud by tedy daný jedinec postupoval ryze synchronně, musel by prvně vytáhnout tšstoviny z obalu a vložit je do vroucí vody. Následně by bylo

nutné vyčkat, než těstoviny dostatečně změknou, aby bylo možné je ze sporáku odstranit a přecedit. Pouze poté je možné vytáhnout pánvičku, vložit ji na uvolněnou plotýnku a začít přidávat ingredience, potřebné na tvorbu omáčky. To samozřejmě není postup, který by byl člověkem automaticky zvolen, má-li k dispozici více než jednu plotýnku.



Obr. 2.10: Synchronní programování

Pokud by tedy jedinec postupoval assynchronně, byly by utilizovány v tomto případě minimálně plotýnky dvě. Jak již bylo řečeno, ty jsou analogií, představující jádra procesoru. V mezičase, kdy se vaří na jedné plotýnce voda s těstovinami, by na druhé plotýnce byla připravována omáčku v pánvi. Omáčka představuje sice druhý krok v tomto postupu, nicméně je její přípravou vážadován kratší čas a tudíž bude tato činnost dokončena dříve, než zdlouhavý první krok, kde není potřeba žádná pozornost.



Obr. 2.11: Asynchronní programování

Naprosto klíčovými jsou v tomto případě příkazy "async"a "await". Async se používá při definici funkce, čímž je kompilátoru sděleno, že se jedná o funkci asynchronní.

Výpis 2.13: Příklad async funkce v Pythonu

```
async def asyncFunction():
```

Příkaz await je posléze použit uvnitř funkce a vyznačuje výsledek operace, který musí být splněn, před dokončením samotné funkce. Tedy pakliže je asynchronně volána jíná funkce, jejíž výsledek je nutný pro možnost pokračování v programu dále, je použito toto klíčové slovo.

2.5.4 Popis programu

Hlavní částí této diplomové práce je samozřejmě lokalizační program samotný. Jelikož se jedná o tak stěžejní část, je nutné vysvětlit jeho jednotlivé části a postupovat podobně, jako tomu bylo v kapitole 2.4.

Launch file

Launch file tohoto programu je poměrně jednoduchý. Je zde možné definovat názvy rámců, které souvisejí s lokalizací. Těmi jsou referenční rámec, obsahující počátek systému, ke kterému je pozice robota vztahována, rámec kamery a dále uživatel má také možnost specifikovat jméno navazujícího rámce. Tento navazující rámec může představovat například tělo robota a slouží pro výslednou pózu, kdy je uživatelem požadováno, aby byla vztažena k jinému bodu, než je kamera. Pakliže žádný takový rámec neexistuje, je uživatel instruktován pomocí poznámky, že je nutné, nechat pole navazujícího rámce prázdné. Výsledná póza je v tomto případě vypočtena mezi počátkem a kamerou. Je tedy pouze na uživateli a jeho definice transformačního stromu, jaká možnost bude využita. Program je navržen tak, aby vyhověl všem požadavkům a možným změnám v rámci jiných aplikací, a nebyl tak použitelný pouze pro lokalizace s podobnou strukturou, jako tomu je při pokusech v rámci této práce. Robot, na který je tento program apliikován, již disponuje vlastním algoritmem pro určování lokalizace. Za účelem koexistence obou algoritmlů zároveň, nebylo žádoucí, odesílat transformaci o pozici z tohoto programu do topicu /tf. Launch file je tak obsažena proměnná "broadcast_tf", kde si uživatel může zvolit, zda informace rozesílat do transformačního stromu a všem jiným běžícím programům nebo ne. Posléze je nutné specifikovat kovarianční matici. Ta je statitická pro všechna měření a odkazuje na nejhorší možný případ, plynoucí z měření statistických veličin, který je popsán v kaptiole 2.5.2. Předposledním parametrem, je nutné nastavit mód samotné detekce. Program obsahuje tyto módy dva - nearest (nejbližší) a average (průměr). Oba tyto módy budou popsány dále v této kapitole. Posledním parametrem, pomocí které je možné lokalizaci napasovat na již vytvořenou strukturu, je jméno výstupního topicu, obsahující zprávy PoseWithCovariance. Program je poté možné spustit pomocí příkazu

Výpis 2.14: Příklad spuštění lokátoru

```
ros2 launch aruco_ros_locator locator.launch.py
```

Setup

Hned po spuštění, je vytvářen subscription (odběr zpráv), který naslouchá topicu "/marker_publisher/markers", kam jsou stejnojmenným programem knihovny aruco_ros posílány zprávy typu MarkerArry, obsahující zprávy Marker. V těch jsou uloženy informace jako například jsou: jméno rámce značky, jeho pozice v obraze kamery a nebo důvěryhodnost detekce.

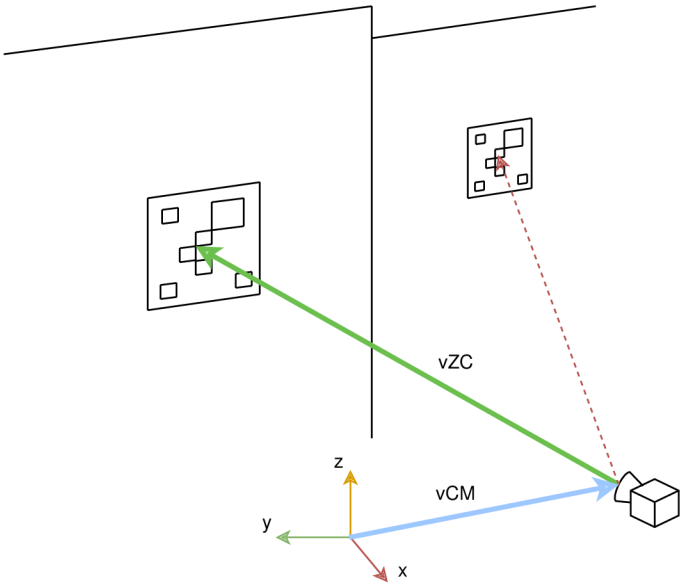
Výpis 2.15: Příklad tvorby subscription

```
self.marker_listener = self.create_subscription(MarkerArray,
'/marker_publisher/markers',self.markerListener_callback, 10)
```

Důvěryhodnost je parametrem, který musí být určován v průběhu zpracovávání obrazu a značky samotné. Bohužel marker_detector neposkytuje reálnou důvěryhodnost a mimo ní, je automaticky nastavena její hodnota na jedničku. Výsledný program této diplomové práce je spíše koncipován ke koexistenci, spolupráci nebo také jako nádstavba aruco_ros, a tedy nebylo možné jakýchkoliv úprav v kódu aruco_ros knihovny tak, aby byl parametr počítán. Od použití této hodnoty dále bylo tak upuštěno. K samotné lokalizaci je ale především potřeba názvu rámce a pozice v obraze. Oba tyto parametry jsou ze zprávy dostupné. Program očekává zprávy tohoto typu a pakliže je zpráva detekována, je zavolána funkce "markerListener_callback". Nyní je způsob zpracování rozdělen na tři možnosti, jakými je algoritmus vykonáván. Který způsob je zvolen, je rozhodnuto pomocí dvou faktorů. Prvním parametrem je momentální počet značek, nacházejících se v obraze. Pokud je v obraze nalezena pouze jedna značka, je její pozice v obraze převedena na pozici kamery vůči značce a algoritmus je tímto ukončen. Pakliže se v obraze ale nachází značek vícero, záleží pak na parametru druhém, kterým je vybraný mód.

Lokalizační módy

Výše v textu bylo zmíněno, že program je schopen pracovat ve dvou různých módech. Tím prvním je "nearest". Pakliže byl tento tento mód uživatelem zvolen, je všemi značkami nacházejících se v obraze proiterováno a je zkontrolována jejich vzdálenost v ose Z, tedy vzdálenost v ose kolmé na rovinu značky. Hodnota Z osy jednotlivé značky je ukládána do proměnné pouze, pakliže je menší než vzdálenost, která je uložena v proměnné v daný moment. Stejnému způsobu změny podléhá i název rámce a pozice v obraze. Pakliže tedy je vzdálenost nové značky menší, než stávající nejnižší vzdálenost, je změněna tato proměnná, pozice a název, které jsou následně brány jako výchozí. Algoritmus končí, jakmile bylo proiterováno všemi značkami v obraze a výstupem je nejbližší značka, jejíž pozice v obraze je následně určitými operacemi převedena na pozici kamery vůči značce.



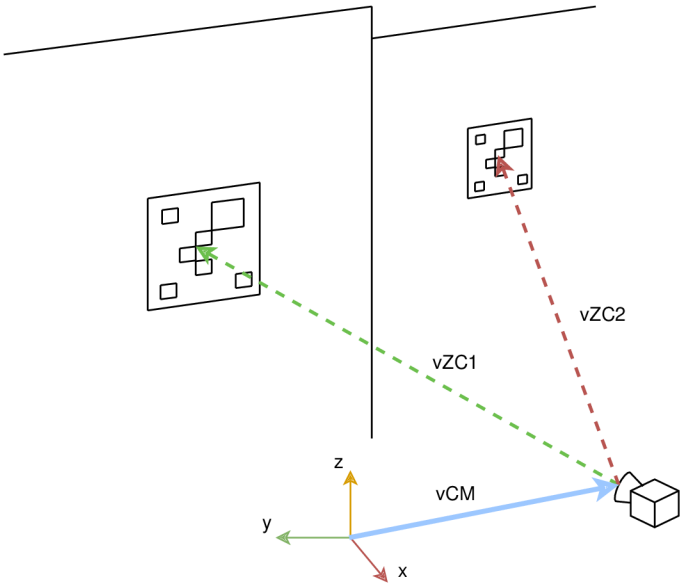
Obr. 2.12: Mód nearest

Na nákrese výše je vyobrazena vazba kamery a značky - vZC, ze které je odvozena konečná vazba kamery vůči počátku - vCM. Vazby vyjadřující pózu značky vůči počátku jsou z důvodu přehlednosti vynechány. Jak je vidět, transformační strom je propojen pouze jednou vazbou, ikdyž programu jsou dostupné dvě. Druhá vazba, která je programem kvůli vyšší vzdálenosti v ose Z ignorována, je značena čerchovanou čárou.

Při módu "average"je taktéž iterováno skrze všechny značky, nicméně namísto hledání nejnižší hodnoty Z, je pro každou značku prvně provedena transformace a je spočítána póza kamery vůči počátku, jako by se jednalo o jedinou značku v obraze. Velikosti ve všech osách jsou následně sčítány napříč všemi iteracemi a výsledkem jsou hodnoty všech lokalizací v ose X,Y i Z. Podobně tak jsou uchovávány veškeré rotace vztažené k počátku, vyjadřené v quaternionech. Pro tento účel existuje pole, které je pro první iteraci prázdné a s každou další iterací, je přidán na konec odpovídající quaternion Po těchto iteracích je vypočítán průměr translace i rotace a pokud si uživatel přeje, je pozice kamery rozeslána vše, programům skrze /tf s vazbou kamery k počátku.

Způsob se zdá být nepřiměřeně složitý, nicméně je dodržení tohoto postupu nutný. Pakliže by program například pouze sčítal hodnoty os a nepřeváděl je na vazby mezi počátkem a kamerou, znamenalo by to, že veškeré značky musí v prostoru být naskládány na jednom místě. To ale není fyzikálně možné, ani požadované. Pro příklad si představme dvě značky. Ty se nachází na dvou rozdílných souřadni-

cích tak, aby bylo možné je sledovat zaráz v jednom záběru kamery. Tím, že jsou umístěny na dvou rozdílných lokacích, je samozřejmě jejich vzdálenost ke kameře rozdílná a tak stejně je rozdílná i rotace. Tyto hodnoty lze sice průměrovat stejným způsobem, jako se průměrují translace i rotace v módu nyní, kamera ale tím ztrácí vazbu k jakémukoliv fyzickému rámci, ke kterému má být tato transformace vztahována. Ve výsledku je tak nutné prvně provést transformaci a teprve poté je průměr možný.



Obr. 2.13: Mód average

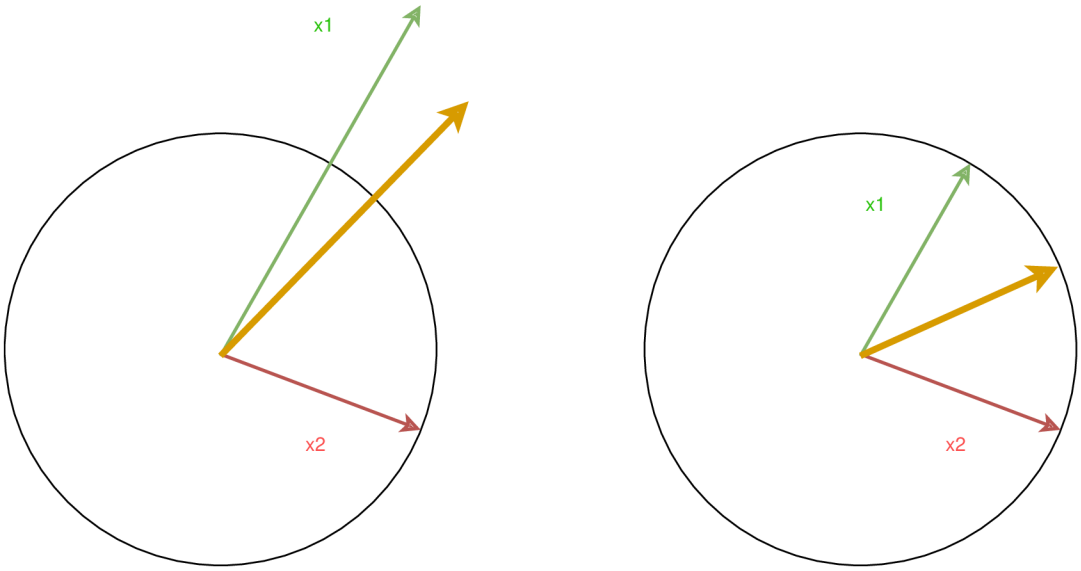
Je možné si z nákresu povšimnout, že v tomto případě je situace značně rozdílná, než tomu bylo u prvního popisovaného módu. Program má k dispozici dvě vazby vCZ, značeny jako vCZ1 a vCZ2. To naznačuje, že jsou obě vazby brané vpotaz. Nicméně obě vazby jsou vyobrazeny čerchovanou čarou. Tyto čerchované čáry značí fakt, že kamer ke značkám ztratila svou vazbu a že je tento rámec již přímo vztahován vůči počátku prostoru.

Průměrování

Mód average (průměr) je ale typem, kvůli jehož funkcím je vyžadována vlastní podkapitola. Průměrování probíha poté, co jsou dokončrny veškeré iterace skrze všechny značky, a tedy jsou k dispozici součty všech os. Průměrování translací je jednoduché. Program místo součtu os obsahuje navíc pole quaternionů, jejichž délka vyjadřuje počet značek nalezených v obraze. Je tak pouze nutné tyto součty podělit délkou pole a výsledná hodnota, je průměrem transformace v určité ose.

Větší problém ale představují průměry rotací. Quaternion je čtyřdimenzionální číslo,jehož vlastnosti nejsou stejné jako tomu je u jednodimenzionálních, reálných čísel. Součtem příslušných složek všech quaternionů a jejich následné podělení počtem, může vyústit v naprosto odlišnou rotaci, která ale není odpovídající její průměrné hodnotě. Existuje mnoho způsobů, jak jednotlivé quaterniony průměrovat. Pro tuto dipolomovou práci byl však použit ten nějjednodušší, nejrychlejší algoritmus - přímé průměrování. Použitá metoda se řadí mezi ty méně přesné, jelikož se jedná pouze o odhad výsledné rotace. Tento fakt ale npředstavuje tak velký problém, jako se může zprvu zdát. Quaterniony, které program průměruje, vyjadřují rotaci svíraného úhlu mezi kamerou a počátkem. V absolutně perfektním případě by tak jednotlivé quaterniony měly naprosto totožné hodnoty a vyjadřovaly by tak stejnou rotaci. Reálně ale existují mezi výslednými rotacemi mírné odchylky. Odhad výsledného quaternionu je tedy v tomto případě naprosto postačující, jelikož jednotlivé rozdíly a tedy výsledná chyba je tak nepatrná, že použití složitějšího algoritmu by bylo zbytečné, či dokonce nežádoucí kvůli delšímu času, který by byl potřeba na zpracování této funkce.

Přímé průměrování quaternionů sdílí určité podobnosti s klasickým průměrování reálných čísel. Stejně tak jako u reálných čísel, jsou sčítány hodnoty jednotlivých os a následně jsou výsledky děleny jejich počtem. Rozdílem je, že je nutné zaručit, aby měly všechny quaterniony stejnou velikost. Quaterniony jsou tak převedeny na jednotkové, jejichž velikost je jedna a jejichž konec leží na trojrozměrné, jednotkové kouli. Pokud by toto zaručeno nebylo, průměr by byl ovlivněn koeficienty násobků jednotlivých jednotkových vektorů. Uvažujme bod v trojtozměrném prostoru, který je středem jednotkové koule. Tento bod je také ohniskem vektorů, které směřují na náhodná místa v daném prostoru s délkou x a rotací w. Pokud mají být nyní tyto vektory spárovány a má být vypočítán vektor jejich průměru, bude vždy převládat vektor s větší délkou x_1, oproti vektoru s kratší délkou x_2 a stane se v tomto průměru dominantním. Výsledné vektory tak budou mít nejen rozdílnou délku, ale také budou mít rotace, které neodpovídají průměrné rotaci jednotlivých párů vektorů.



Obr. 2.14: Průměrování rotací

Je tedy vypočítán jednotkový quaternion a jeho jednotlivé složky jsou sečteny a zprůměrovány tak, jako je tomu u reálných čísel. Po výpočtu průměru je quaternion převeden opět na jednotkový a výsledná hodnota představuje onen průměr.

Lokalizace

Ve všech případech, tvoří jádro detekce následující funkce.

Výpis 2.16: Výpočet vazby

```
tMarker = await self.get_transform_from__marker_pose(pose0)
```

Ta je koncipována tak, aby bylo její použití co nejpružnější a mohla tak být využita na řadu různých případů s rozdílnými potřebami. Funkci lze spustit pouze s parametrem, určujícím pózu značky v obraze. Mimo to je ale možné také specifikovat, zda se návratová póza vztahuje ke kameře a nebo navazujícímu rámci, zda je žádoucí vysílat transformace do /tf a nebo je také možné specifikovat vlastní rámce, mezi kterými je transformace počítána. Zvláště důležitým parametrem je právě rozesílání zpráv do /tf. To je samozřejmě uživatel schopen libovolně měnit v launch file. Pokud je ale použit mód "average", kdy je tato funkce volána v každé iteraci pro transformaci vazby kamera-značka na vazbu počátek-kamera, je žádoucí mít možnost tuto funkcionalitu zcela zakázat, nehledě na uživatelská nastavení.

Po vytvoření zprávy, kterou tato funkce produkuje, jsou nastaveny rámce, mezi kterými je transformace počítána. Tato zpráva bude nyní použia pro doplnění rámce kamery do stromu transformací a propojení tak všech potřebných vazeb. K tomu je

využita právě póza značky v obraze. Jedná se ale prakticky o vektor směřující od kamery ke značce. Je tedy nutné otočit jeho směr pouhým vynásobením souřadníc číslem -1. Operace s vektorem však nejsou u koce. Ten má nyní sice správný směr, ale je nadále nutné zakomponovat samotné rotace do translačního vektoru.

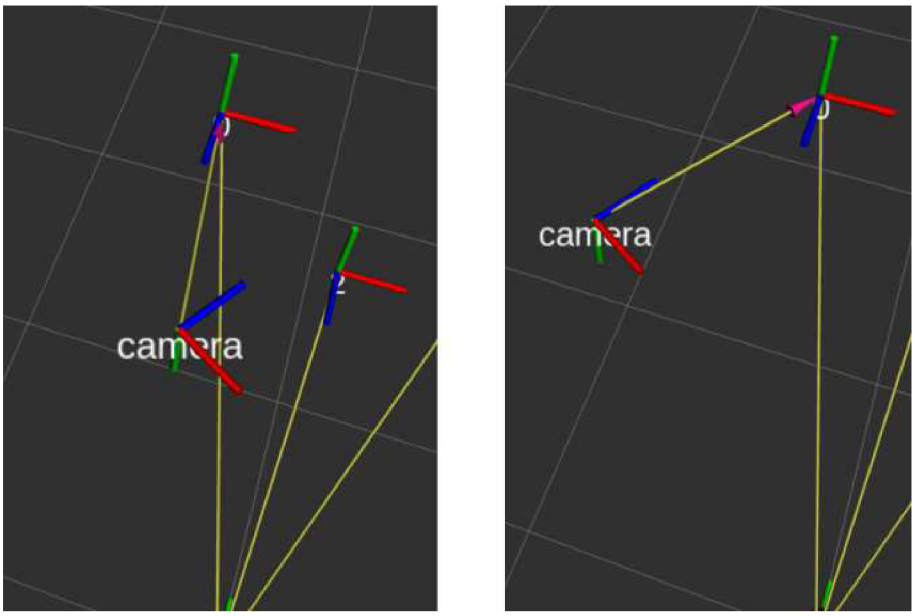
Výpis 2.17: Přepočet vektoru

```
_t.transform.translation = Vector3(x=marker_pose.position.x,
 y=marker_pose.position.y, z=marker_pose.position.z)
_t.transform.rotation.x = -marker_pose.orientation.x
_t.transform.rotation.y = -marker_pose.orientation.y
_t.transform.rotation.z = -marker_pose.orientation.z
_t.transform.rotation.w = marker_pose.orientation.w
_t = self.rotateTVec(_t)
```

Doposud bylo s translací a rotací značek zacházeno jako by se jednalo o samostatné entity. To je ale samozřejmě nesmyslné. Rotace i translace jsou složky většího celku, nazývaným jako póza. Tyto složky jsou spolu úzce spjaty a je jedna druhou vzájemně ovlivňována. Právě funkce "rotateTVec"je zodpovědná za již zmiňovné upravení translačního vektoru tak, aby odpovídal své vlastní rotaci. Hlavní složkou této metody je tzv. rotační matice. Matice je v podstatě aplikace rovnice 1.10 z teoretické části této práce, která je reprezentována maticí 3x3, kvůli nutnosti inkorporace této rotace do trojdimenzionálního vektoru translace. [41] Její výpočet je značně zdlouhavý, nicméně výslledkem je obecná matice, kterou lze přímo a jednoduše použít v kódu, bez nutnosti dalších počtů. Rotační matice je definována jako:

$$\begin{bmatrix} 1-2\cdot(y^2+z^2) & 2\cdot(xy-wz) & 2\cdot(xz+wy) \\ [2\cdot(xy+wz) & 1-2\cdot(x^2+z^2) & 2\cdot(yz-wx) \\ 2\cdot(xz-wy) & 2\cdot(yz+wx) & 1-2\cdot(x^2+y^2) \end{bmatrix} \tag{2.1}$$

Výsledený translační vektor je následně získán skalárním součinem vstupního vektoru translace a rotační matice, sestavené ze složek quaternionu.



Obr. 2.15: Lokalizace bez rotační matice vs s maticí

Póza z této funkce je vkládána do stromu transformací a je vytvářen rámec kamery. Záleží pouze na požadavku uživatele, zda bude informace o transformaci rozeslána všem uzlům a nebo bude vazba dostupná pouze v rámci programu. Pakliže se jedná o případ, kdy má být transformace rozesílána, je použit příkaz sendTransform z nabídky funkcí funkcí tf_boradcasteru. Pokud se ale jedné o případ druhý, je použita funkce samotného tf_bufferu. Tf_buffer úzce spolupracuje s tf_listenerem a mimo jiné je v jeho lokální paměti (myšleno paměť v rámci jednoho programu) udržován celý strom trnasformací. Tohoto faktu lze využít a vytvořit tak vazby mezi objekty, které ale prakticky mimo rámec programu neexistují, jelikož o těchto vazbách nebyly žádným jiným uzlem přijmuty jakékoliv informace.

Výpis 2.18: Tvorba lokální vazby

```
self.tf_buffer.set_transform(_t, 'default_authority')
```

Poté je na řadě poslední, neméně důležitá část detekce. Tou je vyčítání. Pro rekapitulaci, do funkce byla poslána póza značky v obraze. Ta byla upravena na hodnoty, aby odpovídala vektoru směřujícímu od značky ke kameře. Následně byl tento vektor vložen do stromu transformací a tím i vytvořen rámec kamery. V tento moment je tedy pouze nutné pózu mezi nulovým bodem rámce map a kamerou či podřadným rámcem, vyčíslit. Transformace byla sice do stromu poslána, nicméně není možné její hodnoty vyčítat okamžitě. Mezi momentem, kdy je transformace do stromu poslána a momentem, kdy je samotný strom aktualizován, existuje prodleva, která má délku nižšího počtu násobků frekvence volání tf_bufferu. Jelikož není tato

prodleva přesně definována a mimo jiné záleží i na výpočetním výkonu použitého hardware, je nutné se na výslednou transformaci neustále doptávat a pouze tehdy, kdy je póza vrácena bez jediné chyby, je možné pokračovat dále. Tento způsob vyčítání je jedním z hlavních důvodů použití asynchronního programování. Samotný tf_buffer obsahuje funkci, která byla vytvořena přesně za účelem splnění tohoto úkonu. Funkce je ale definována jako asynchronní s návratovým typem future neboli proměnná, která bude doplněna v budoucnu. Transformace, kterou program očekává je vazou mezi počátkem a kamerou. Pro případ, kdy nastane chyba, je definován timeout na výslednou pózu, který je nastaven na jednu sekundu a při překročení této hodnoty je vyčíslení v daný moment ukončen a program čeká na následující iteraci. Při testování ale nebylo nezaznamenáno, že by podobný scénář nastal. V opačném případě je obdržena informace o tom, že rámec kamery opravdu existuje a je zavolána funkce "lookup_transform"mezi počátkem a kamerou či podřadným rámcem. Tato transformace vyjadřuje konečnou pózu specifikovaného rámce od nulového bodu mapy.

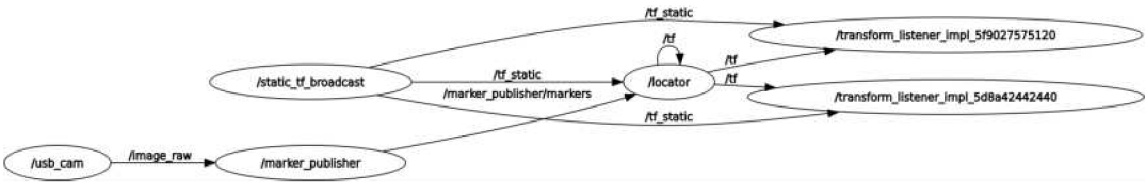
Výpis 2.19: Tvorba lokální vazby

```
getTransform_future = self.tf_buffer.wait_for_transform_async(
_toFrame, _fromFrame, rclpy.time.Time(nanoseconds=0))
asyncio.wait_for(getTransform_future, timeout=1.0)

if getTransform_future.done():
    try:
        _t = self.tf_buffer.lookup_transform(_fromFrame_ret,
        _toFrame_ret, rclpy.time.Time())
    except TransformException as ex:
        self.get_logger().info(Could not transform
        from{_fromFrame_ret} to {_toFrame_ret}: {ex})

return _t
```

2.5.5 Celkové shrnutí

Konečný balíček tedy obsahuje tři programy. Jedním z nich je možné naplnit strom transformací, pomocí jednoduchého setup souboru, obsahující jednotlivé translace a rotace značek ve vztahu k referenčnímu rámci neboli počátku. Jelikož je často žádoucí, aby transformace pracovala určitým způsobem s kovarianční maticí, je součástí balíčku i statistický program, jehož účelem je sestavení právě oné matice. Mimo to, jsou ve výstupním excel soubor specifikované další statistické veličiny a veškeré naměřené pózy v obraze v daném časovém úseku měření. Uživatel

je tak schopen vypočítat z dat libovolné veličiny ke přesnějšímu zkoumání přesnosti detekce. Posledním programem je samotná lokalizace. Ta je koncipována jako nádstavba uzlu marker_detector z knihovny aruco_ros. Od tohoto uzlu je získáváo pole všech značek s jejich názvy a pózami v obraze.Je nutné, aby byly uživatelem nastaveny veškeré potřebné proměnné v launch file včetně volby lokalizačního módu a následně je posílána na výstup zpráva typu PoseWithCovariance, obsahující výslednou pózu kamery (nebo podřadného rámce) vůči počátku.



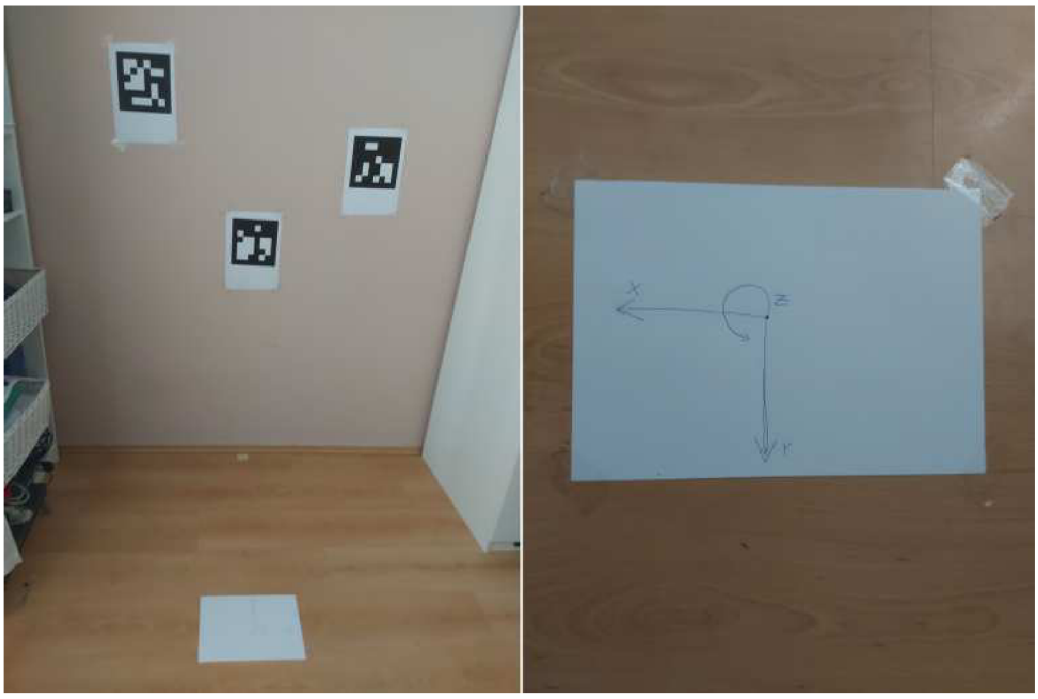
Obr. 2.16: Graf lokalizace

V grafu nad tímto textem je možné si povšimnout, které uzly spolu komunikují a jak. V tomto případě bylo vysílání do /tf zapnuto. Kamerový vstup tedy posílá obraz do uzlu /marker_publisher. Zároveň jsou do tohoto uzlu a do lokátoru posílány informace o statických vazbách, ktereé převážně definují strom transformací. Oba tyto uzly potřebují znát dané vazby, jelikož každý z nich obsahuje svůj vlastní tf_buffer, který si udržuje strom ve své vlastní paměti. Marker_publisherem je produkováno již zmíněné pole značek do lokátoru. Lokátor následně Vytváří ve stromu rámec kamery a je skrze něj dopočítávána vzdálenost od počátku. V tomto případě je vše posíláno do simulačního programu rviz2, který byl zmíněn v kapitole 2.3

2.5.6 Testování

Prvotní testování

Program byl v průběhu vývoje důkladně testován. Na počátku bylo možné značku postavit na stůl a kamerou snímat přibližnou vzdálenost a natočení. S postupem času v rámci vývoje, bylo ale potřeba vytvořit testovací aparát s pevně definovanými vzdálenostmi, aby bylo možné zjišťovat momentální přesnosti lokalizace. K tomu byla tedy využita část stěny v místnosti s přibližnými rozměry 2x3 metrů, na kterou byly nalepeny tři značky s id od 0 až po id 2. Na podlahu, byl do určité vzdálenosti nalepen papír s vyznačeným nulovým bodem, který znázorňuje počátek prostoru. Veškeré vzdálenosti byly změřeny tak, aby se nepřesnost pohybovala maximálně v rámci desetin cm.



Obr. 2.17: Graf lokalizace

Výsledky měření byly následně přepsány do "setup.txt"souboru a ten byl předložen uzlu static_tf_boradcast k vyvtoření transformačního stromu. Setup.txt obsahuje následující vazby.

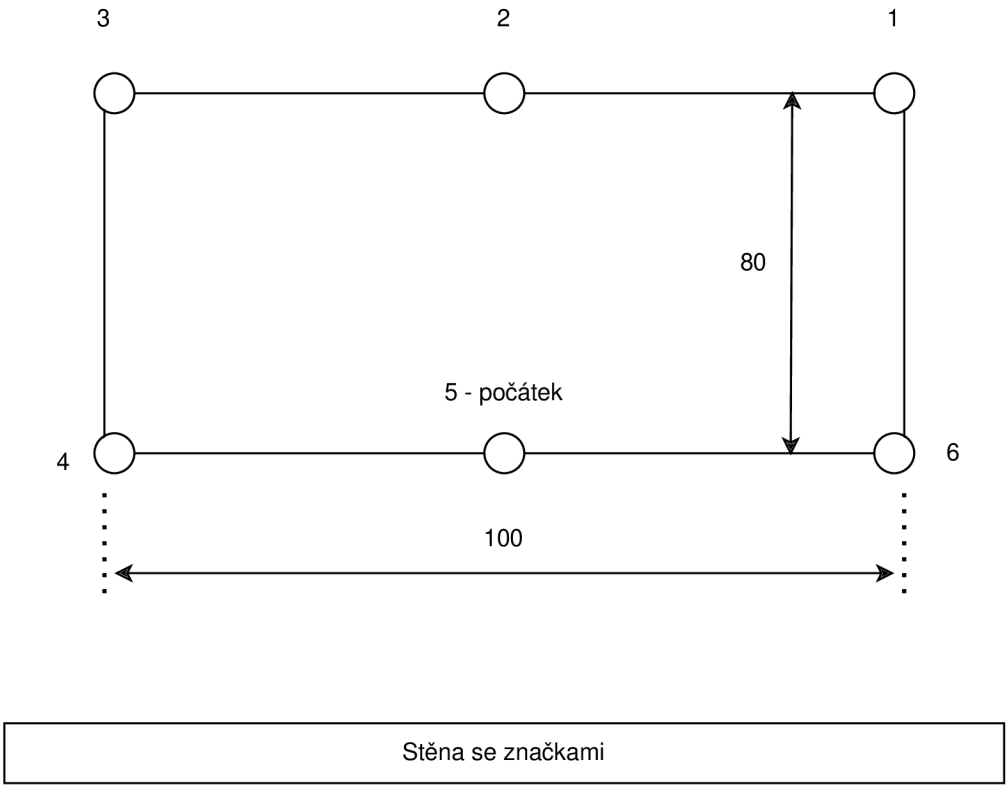
Výpis 2.20: Vazby v setup.txt

```
0 -0.44 0.875 1.47 1.57 0 0                                    1
1 0.37 0.875 1.255 1.57 0 0                                    2
2 0 0.875 0.965 1.57 0 0                                       3
```

Pro připomenutí, struktura tohoto souboru je ['název frame' 'Tx' 'Ty' 'Tz' 'Rx' 'Ry' 'Rz']. Strom byl následně vyobrazen v simulačním programu Rviz2. Na tomto aparátu byl zkonstruován pokus, jehož výsledky by měly co nejlépe napodobit pohyb a kamerový obraz robota. V kapitole 2.5.2 byl sice podobný pokus proveden, nicméně značka se v tomto případě nacházela na středu obrazu s nulovou rotací kamery vůči značce či naopak. To při samotné lokalizaci nebude moc časté, ani vyloženě žádoucí. Při vývoji bylo zjištěno, že kolem nulové osy rotace existuje jisté pásmo necitlivosti. Nulové rotace je při reálném pokusu či aplikaci složité dosáhnout a tedy bude v každé ose často existovat jisté, pro oko zanedbatelné orotování vůči nulovému bodu rotace. Při tak malé chybě je čtverec hran, jehož tvar aruco detektce posuzuje, deformován pouze nepatrně a je tedy pro program složité rozlišit, zda rotace vůbec existuje a pokud ano, jaký má směr. Často se tak stávalo, že při delším sledování simulace, vznikalo okolo této osy určité kmitání, jehož amplituda

se marginálně zvětšuje, při zvyšující se vzdáleností. Při jednom měření byly v takovém případě naměřeny kmity, dosahující až 1.4 metru při vzdálenosti 6.5 metrů od značky. Je tedy žádoucí kameru postavit tak, aby značka, pomocí které se robot lokalizuje, byla v obraze zachycena pod určitým úhlem θ.

V rámci domácího měření byl zkonstruován pokus, jež má úkol ověřit odchylky programu, při vzdálenostech do 80 cm. Vzdálenost není, kvůli omezeném prostoru, vypovídající o celkové robustnosti lokalizace. Lze ale přibližné zjistit přesnosti při malých vzdálenostech. Pro tento účel byly kolem nulového bodu prostoru (počátku) změřeny vdálenosti 50 centimetrů v ose X a 80 centimetrů v ose Y. Z těchto bodů byl následně zkonstruován pomyslný obdélník, po jehož stranách byl využitý notebook s kamerou posouván, V předem určených bodech byly poté změřeny vzdálenosti pomocí lokalizačního uzlu. Tyto vzdálenosti byly měřeny pod různými úhly, aby bylo možné, co nejlépe simulovat kamerový záznam z robota.



Obr. 2.18: Schéma pokusu

Každý bod byl měřen minimálně po dobu pěti sekund a z dostupných póz byly vybrány tři poslední hodnoty. Tato technika byla do měření zahrnuta kvůli možné době ustálení pozice rámce v prostoru. Při měření byla vyvinuta snaha k napozico-

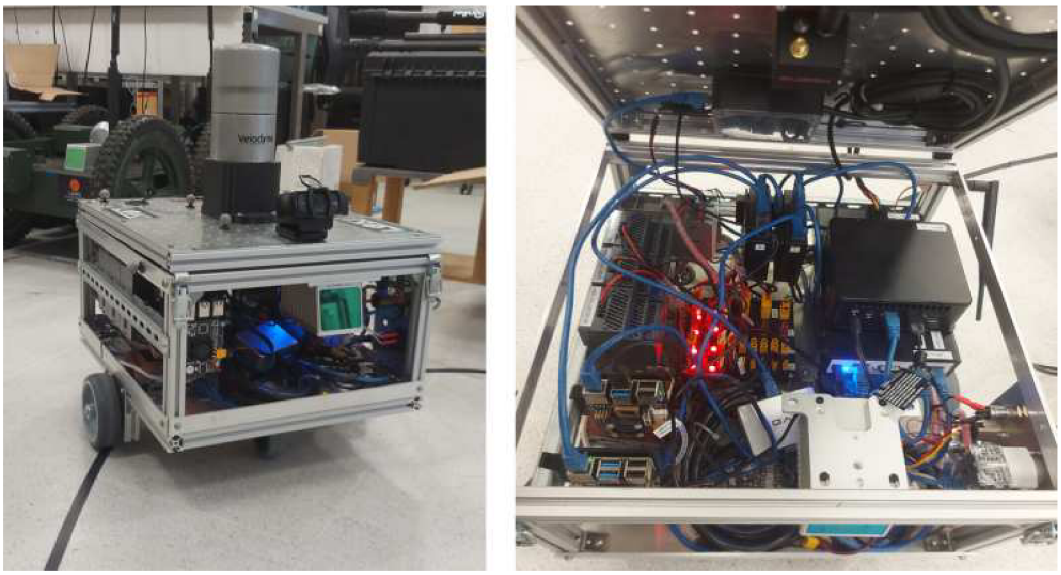
vání notebooku přesně na určené body. Ne vždy to ale bylo možné kvůli omezenému prostoru a nutnosti zahrnout v obrazu kamery co nejvíce značek. Výslekdy průměrných chyb těchto měření jsou vypsány v tabulce níže.

-	x	y	z
1	+2.45	-4.34	-6.69
2	+3.41	+2.59	-7.39
3	-3.87	+9.85	-6.55
4	+0.29	-6.66	-3.35
5	+0,91	-4.46	-2.31
6	+1.42	-4.75	-3.8

Měření při tomto pokusu potvrdilo, že lokalizace je poměrně přesná pro menší vzdálenosti. Bylo by samozřejmě možné dané výsledky nadále vylepšit za předpokladu, že je k pokusu možné použít kvalitnější kameru s lepší filtrací obrazového šumu a ta je důkladně zkalibrována. Tento pokus byl do diplomové práce zahrnut díky rozdílným parametrům méně kvalitní kamery, která byla pro toto měření použita.

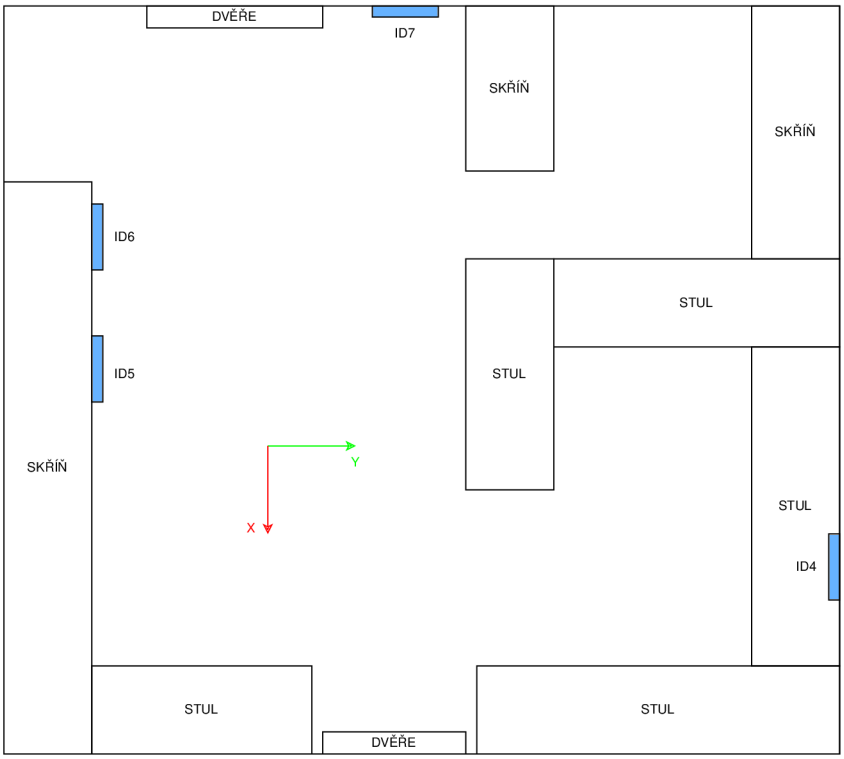
Aplikace na robota

Reálná aplikace vytvořeného projektu na reálného robota, probíhala v laboratoři robotiky na ústavu automatizace. Za asistence vedoucího práce a jeho kolegů. Pro tyto potřeby byl z repertoáru zařízení, vyčleněn robot, nesoucí název Loki.Loki je robot hranatých rozměrů, kterého je možné ovládat, pomocí zapůjčeného konzolového ovladače. Dále je robot obohacen o zažízení Velodyne, což je otočný Lidar senzor, používaný k mapování prostředí. Na tohoto robota byla vedoucím práce připevněna kamera, která byla důkladně napozicována tak, aby měla nulový posun v ose Y na střed horní desky. V ose X pak posun činil 18 centimetrů.



Obr. 2.19: Robot Loki

Samotné měření pak probíhalo čtyři dny. První den byl využit převážně na instalaci a přípravu všech balíčků, které jsou potřeba k výsledné lokalizaci. Na robota bylo možné se připojit skrze síť, za použití SSH protokolu. Centrální zařízení, na které se uživatel (to znamená i já) připojuje, tvoří počítač menších rozměrů, s označením NUC. Na tomto počítači je nainstalován operační systém Linux Ubunntu, takže postup byl stejný, jako tomu bylo u domácí instalace a používání. Po připojení byl tak v mé vlastní dedikované složce vytvořen adresář "ros2_ws", kam bylo potřeba stáhnout veškeré knihovny. Jak již bylo v této práci zmíněno, jedná se hlavně o balíčky aruco_ros, usb_cam a image_pipeline. Nejdelší dobu instalace ale zabrala knihovna OpenCV. K její instalaci bylo po stažení potřeba vyhradit asi hodinu a půl. V mezičase byly na stěny laboratoře naměřeny vzdálenosti a nalepeny značky. Ke zjednodušení této činnosti byly odměřeny rozměry gumových plátů, ze ktré je podlaha složena. Tento plát tvoří ve výsledku čtverec o délce strany rovné 61 centimetrů. Značky byly nalepeny tak, aby jejich střed protínal spoj, mezi těmito pláty a jejich vzdálenost byla skrze ně dopočítána.



Obr. 2.20: Rozložení místnosti

Tyto vzdálenosti byly následně vepsány do "setup.txt", sloužící k rozeslání statických transformací. Soubor měl následující strukturu.

Výpis 2.21: Vazby v setup.txt

```
0 -0.44 0.875 1.47 1.57 0 0 DOPLNIT                                    1
1 0.37 0.875 1.255 1.57 0 0                                            2
2 0 0.875 0.965 1.57 0 0                                               3
```

Počas instalace openCV knihovny byl objeven problém, chybějícího "include"příkazu v jednom ze souborů. Poměrně rychle bylo ale možné chybu dohledat, za dotazů na stránce GitHub, týkající se právě oné knihovny. Tento problém bude blíže specifikován v kapitole 2.5.7. Po dokončení měření i instalace, byly všechny balíčky zkompilovány a jejich launch soubory nastaveny tak, aby bylo možné s robotem začít pracovat. Zde byl ale odhalen první problém, týkající se nerozesílání se statických transformací zařízením v lokální síti. V topicu /tf_static byly sice všechny zprávy k nalzení, nicméně o jejich rámcích nemělo informace žádné ze zařízení. Tento problém se podařilo vyřešit až druhý den, přidáním do programu možnost znovu poslání, pakliže je změněn počet zařízení nebo programů, kterými je danému topicu naslouchán.

Druhý den bylo zaměřeno prvotním testům. Po zapnutí robota, jsou jedním z

jeho uzlů rozeslány statické transformace, které znázorňují jednotlivé jeho části. Příkladem mohou být kola, geometrický střed robota a nebo jeho horní deska. Na rámec právě této desky, pojmenovaného jako "loki_1_top_plate"byla navázána kamera, za použití vyslání statické transformace, díky integrovaného uzlu tf2 knihovny.

Výpis 2.22: Vazby v setup.txt

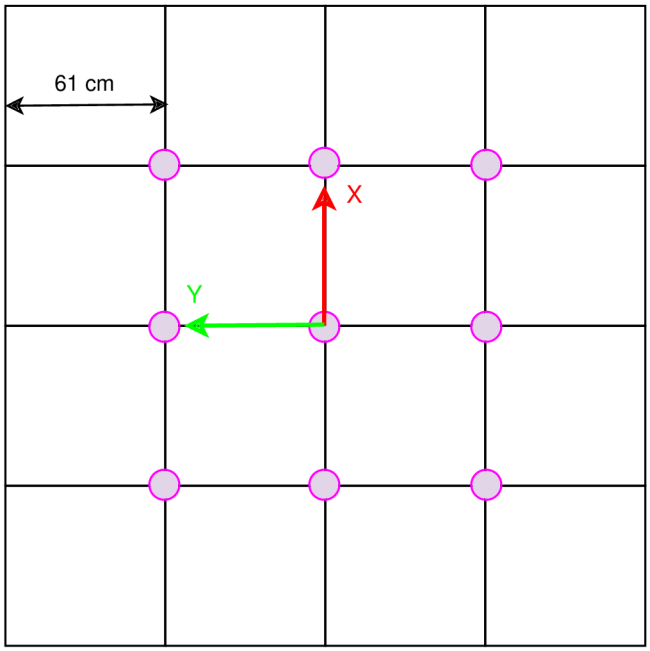
```
ros2 run tf2_ros static_transform_publisher x y z yaw pitch
roll frame_id child_frame_id
```

Robota i se všemi balíčky se následně podařilo vcelku rychle nastavit a mohlo tak být započato měření. Při tom bylo tedy ježděno po místnosti a byly do obrazu kamery zabírány různé značky. Vzdálenosti byly určovvány dle gumových plátu, kam byl robot ńapozicován tak, aby jeho geometrický střed ležel na průsečíku vodorovných a svislých spárů zmíněných plátů, tvořící podlahu. Byly vyzkoušeny oba dva módy a různé vzdálenosti. Při použití "nearest"(nejbližší) módu, se odchylka pohybovala přibližně v rámci 30 až 40 centimetrů. Tato odchylka se ale marginálně zvětšila v moment, kdy byl mód přepnut do "average"(průměr). Toto chování bylo nežádoucí a nesouhlasilo s charakterem měření, které s programem probíhaly do té doby. Prvotní odhad byl, že v kódu se nachází chyba. Ta byla opravdu objevena a týkala se opravdu výpočtu transformací, při použití tohoto módu, kdy nejsou vysílány transformace do topicu /tf. Část kódu, která má na starosti posílání zprůměrované pozice rámce kamery zpět do stromu, nebyla pro případ, kdy transformace rozesílány nejsou, dostupná. Po opravě chyby, byly ale odchylky stále moc vysoké a měření bylo tedy ukončeno.

Příčinou této nepříjemnosti byla chyba, která nebyla při testování plně odhalena. Jednalo se totiž o chybu, projevující se pouze ve specifickém prostředí laboratoře. Tam byl vytvořen strom, do kterého volně přispívalo jakýkoliv zapnutý robot vysílal skrze určité algoritmy svůj vlastní strom, popisující jeho strukturu a lokaci. Zde nastal onen problém. Výsledný program této diplomové práce totiž potřebuje pro svou funkci, aby strom obsahoval pouze počátek, značky a případně jakékoliv rámce, nesouvisející s aruco lokalizací. Program po získání pózy značky v obraze, vytvoří vlastní rámec, který je poté rozesílán do stromu transformací, nebo je ukládán v paměti tf_bufferu v rámci programu. Která z těchto možností je zvolena, zavísí na nastavení launch file, jako to bylo popsáno v kapitole 2.5.4. V případě tesotvání na aparátu v laboratoři, byla transformace uchovávána pouze v rámci paměti tf_bufferu, jelikož jak už bylo v této práci zmíněno, robot obsahuje svůj vlastní lokalizační systém, orientující se na základě odometrie, neboli počtu otočení kol. Po přijetí pózy značky v obrazu kamery, je následně vytvořen rámec kamery, případně i navazujícího rámce a ta je vložena do stromu. Následně je zavolána funkce "lookuptransform", která s dotyčným tf_bufferem komunikuje a zjišťuje lokace jednotli-

vých rámců. Jelikož ale byla lokace z aruco značek, ukládána pouze lokálně a globální proměnná rámce kamery i navazujícího rámce existovaly ve stromu, díky odometrii, byl při každé aktualizaci stromu, v paměti tf_bufferu tato hodnota přepsána a program vypisoval hodnoty, poskytnuté odometrií. Důvod rapidního zhoršení lokalizace, při změně módu, plynulo pravděpodobně z restartu robota, kdy následně nebyl spuštěn z nulové pozice, a tedy byla jeho chyba v lokaci výrazně vyšší.

Byly tedy vytvořeny nové rámce "aruco_camera"a "loki_1_top_plate"a mezi nimi byla definována stejná transformace, jako tomu je u reálného modelu. Následně bylo s robotem ježděno po místnosti a kamera byla natočena směrem k libovolným značkám. Pro ulehčení práce, byl robot vždy postaven na průsečík hran gumových čtverců se známou velikostí, jako to bylo popsáno výše v textu.



Obr. 2.21: Průběh měření

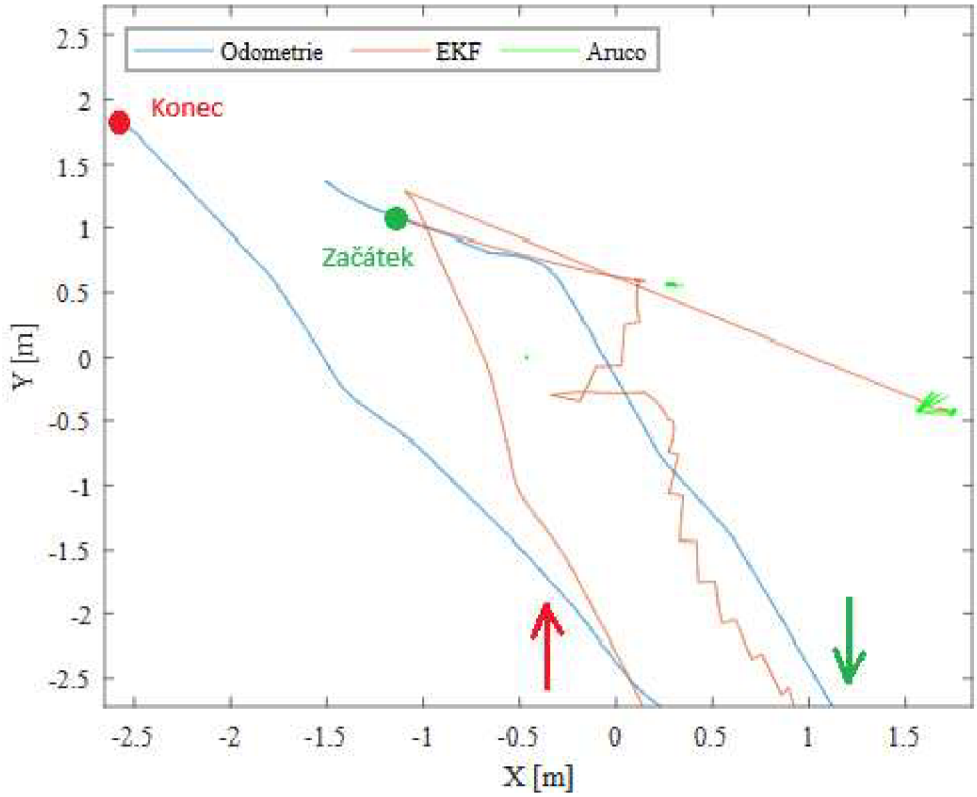
V rácmci měření byly otestovány oba dva módy a jejich lokace byla měřena ze stejných bodů v prostoru, jako je vidět na nákresu výše. Následně byly ke každému měření přidány tři náhodné body, které měli za úkol docílit nejsložitější lokalizace, a tedy nejhorších výsledků. Všechna měření byla opakována minimálně třikrát a z tšchto měření byla brána průměrná hodnota. Výsledky těchto měření jsou uvedeny v tabulce níže.

Reálná lokace			Nearest mode			Average mode		
x	y	z	$\hat{x}$	$\hat{y}$	$\hat{z}$	$\hat{x}$	$\hat{y}$	$\hat{z}$
0.00	0.00	0.20	0.057	0.034	0.271	0.094	0.078	0.276
0.00	-0.61	0.20	0.02	-0.585	0.285	0.027	-0.518	0.258
-0.61	-0.61	0.20	-0.583	-0.632	0.273	-0.631	-0.496	0.255
-0.61	0.00	0.20	-0,628	0,078	0,231	-0.611	0.042	0.271
-0.61	0.61	0.20	0.6	0,535	0.211	-0.536	0.732	0.29
0.00	0.61	0.20	-0.008	0.645	0.2365	-0.022	0.744	0.218
0.61	0.61	0.20	0.523	0.598	0.175	0.558	0.789	0.210
0.61	0.00	0.20	0.551	0.038	0.178	0.655	0.189	0.223
0.61	-0.61	0.20	0.594	0.489	0.278	0.651	-0.435	0.234
Error			1.502	1.418	0.433	0.377	1.125	0.435

Jak je možné na hodnotách vidět, chyba se ve většině případů pohybuje v rámci jednotek centimetrů. Ikdyž není přesnost dokonalá, jedná se o lepší výsledky, než bylo zprvu očekáváno. Obecně je také z měření možné vyvodit závěr, že mód "AVERAGE"je vpodstatě o něco málo přesnější, než tomu je u módu "NEAREST". Tento výsledek byl jistým způsobem neočekávaný. Jelikož tento mód bere vpotaz veškeré značky, které se v daný moment v obrazu nachází, je lokalizace nejbližší značky (která by měla mít i nejlepší výsledky) zhoršována výsledky, plynoucí z hůře detekovatelných značek. V daném prostoru, kde byl ale robot testován, se nacházely tyto značky velmi blízko sebe, a tedy nebylo zhoršení v lokalizaci natolik marginální. Hlavní výhodou tohoto módu je jeho robustnost vůči šumu. Právě tomuto pozitivnímu efektu průměrování je připisována vyšší přesnost. Efekt plyne z faktu, že pakliže je šum čistě náhodný, pak jejich momentální směr může u dvou různých značek, nabývat protichůdných směrů, jejichž přínos je tak vyrušen.

Jak již bylo v této práci zmíněno, pozice robota byla do této chvíle odhadována pomocí odlišných metod, zabývající se problematikou lokalizace předmětu v prostoru. Jako finální test byl tak balíček aplikován do tohoto systému a byla ověřena jeho kooperace s jinými metodami. K tomu byl použit Kalmanův filtr z knhovny robot_localization. Filtr následně využívá kovariančních matic, dostupné ze zpráv typu PoseWithCovariance, jež jsou poskytovány mnou vytvořeným řešením. Mimo to musí být samozřejmě sestaveny matice P, Q, R, které však nebylo nutné sestavovat, jelikož jejich forma již byla specifikována. Zbývalo tedy pouze propojení těchto dvou knihoven. K tomuto účelu, existuje v robotu parametrizační soubor, obsahující mimo jiné topicy, které jsou napojeny na vstup filtru. Po úpravě souboru a restartu robota, bylo možné přejít k finálnímu měření. Robot byl spuštěn přesně v nulovém bodě a je natočen souhlasně s osou X. Tento krok je nutné dodržet právě kvůli odo-

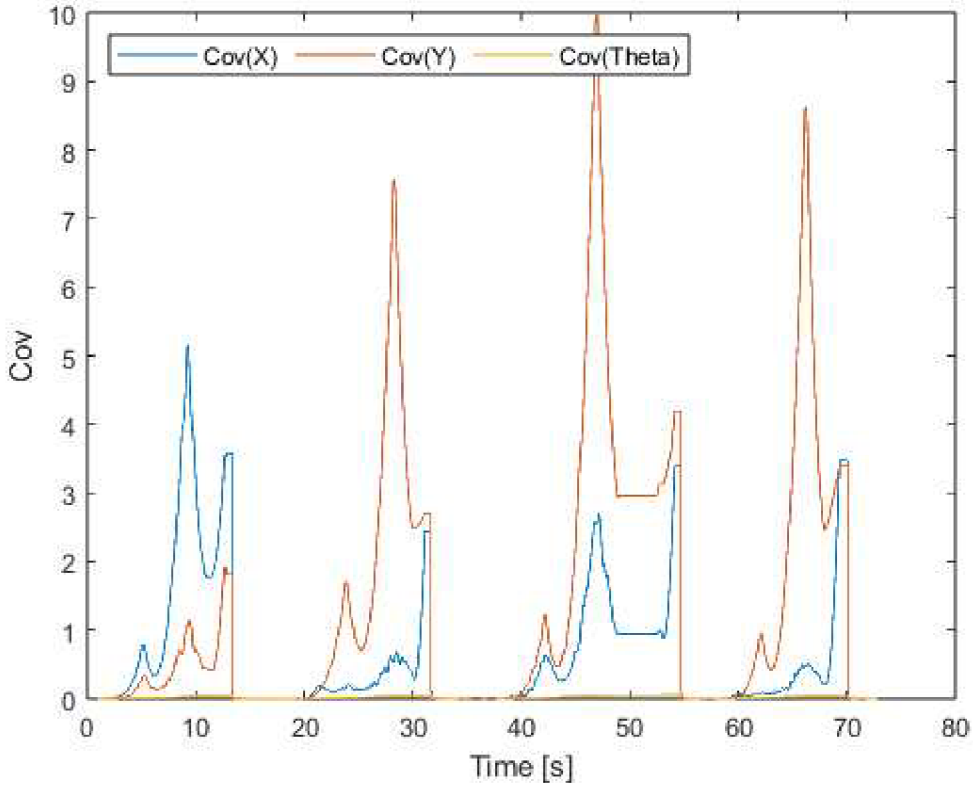
metrii, pro kterou je vyžadována přesná znalost startovní pozice. Bylo již zmíněno, že odometrie není robustním systémem vůči chybě lokalizace a není ani schopen, chybu jakkoliv minimalizovat. Chyba, která je například způsobena protáčením kol, vzrůstá společně s ujetou vzdáleností robota. Není tak neobvyklé, že po pár minutách ježdění, je velikost odchylky pozice v řádu metrů. Ke korekci chyb jsou následně využity značky aruco a mého programu.



Obr. 2.22: Průběh finálního měření

Na grafu výše, je vykreslen průběh tohoto měření. S robotem bylo ježděno hlavně po chodbě, kde byla snaha, co nejvíce naintegrovat chybu. To mělo za následek zhoršení hodnot kovarianční matice. Ve chvíli, kdy byla chyba dostatečně vysoká, bylo vjeto s robotem zpátky do místnosti, kde byla pozorována skoková změna jeho pozice, což bylo následkem korekce chyb, při zpracování značky a vypočítání pozice. Je vidět, že výsledek EKF (jež je spojením těchto dvou metod a představuje tedy finální pozici robota), je ze začátku přibližně souhlasný s odometrií. Zde ještě robot neurazil takovou vzdálenost a kovarianční matice, znázorňující předpokládanou chybu, obsahuje nízké hodnoty. Po návratu zpět do místnosti, má již kovarianční matice sice špatné výsledky, nicméně filtru není poskytována žádná jiná reference, a tedy je jejich průběh stále souhlasný. V momentě, kdy ale kamera robota zachytí

značku, je tato hodnot pozice předána filtru, s kovarianční maticí, obsahující řádově nižší čísla. V tento moment je tedy pozice značky brána jako dominantní a pozice je skokově opravena. Tento fakt je také vidět, v grafech hodnot zmíněné matice.



Obr. 2.23: Průběh finálního měření

Jak je v tomto příkladu vidět, kovarianční matice filtrované lokalizace, je skokově snižována, v závislosti na přítomnost značky v obraze.

2.5.7 Postup instalace

Při aplikaci na robota, bylo nutné balíček a všechny jeho potřebné části nainstalovat znovu. Při tom se ale objeveny nové typy problémů, které se při první instalaci neprojevily. Tato kapitola tak přesně reflektuje tuto zkušenost.

Prvním krokem je stažení knihovny OpenCV, které je využíváno knihovnou aruco_ros, usb_cam a jinými. V instalačním souboru aruco_ros je specifikované, že OpenCV musí mít přesnou verzi 4.2.0. Nikde jinde bohužel tato informace k nalezení není, tudíž je potřeba být nanjevíě obezřetný při tomto kroku. Po stažení je potřeba

knihovnu nainstalovat na operační systém, pomocí příkazů "cmake"a "make". Instalace by měla zabrat přibližně hodinu. Před samotnou instalací je ale nutné vyřešit ještě jeden problém se samotnou knihovnou. Pokud se uživatel bude snažit knihovnu nainstalovat bez její opravy, instalace může selhat při 80 procentech s následující chybovou hláškou.

Výpis 2.23: Chyba instalace OpenCV

```
 'sleep_for' is not a member of 'std::this_thread'                    1
 456 | std::this_thread::sleep_for                                    2
(std::chrono::milliseconds{2});                                       3
```

pakliže tato okolnost nastane, je potřebné otevřít soubor "gapi_async_test.cpp"a na začátek přidat řádek.

Výpis 2.24: Chyba instalace OpenCV

```
#include <Thread>                                                     1
```

Tímto by měl výt problém vyřešen a je možné instalaci spustit znovu. Ta bude pokračovat od předem dokončených 80 procent.

Druhým krokem je stažení všech potřebných knihoven, které jsou potřeba. Jedná se o knihovny *aruco_ros, usb_cam a image_pipeline (knihovny označené '*' jsou vyžadovány, ostatní nejsou nutně knihovny, které musí uživatel používat, ale byly využity v tomto projektu). Po stažení je nutné vložit veškeré knihovny do ros2_workspace a všechny knihovny zkompilovat, pomocí příkazu.

Výpis 2.25: Instalace knihoven

```
colcon-build --symlink-install                                        1
```

Většina parametrů by měla být nastavena tak, aby bylo možné je spustit hned po nainstalování. Je ale nutné provést kalibraci a ve výsledném souboru specifikovat, že se rámec kamery nazýva "camera", nebo změnit nastavení launch file lokalizátoru, kde je možné přespecifikovat název rámce kamery. Taktéž je nutné zkontrolovat, že obraz kamery je opravdu posílán do topicu /image_raw a informace o kameře do topicu /camera_info.

Dále je uživatel nucen určit nulový bod a změřit vzdálenosti značek od tohoto bodu. Výsledné vzdálenosti je posléze potřeba vepsat do textového souboru "setup.txt", který se nachází ve složce "your_ros2_workspace/src/aruco_ros_locator/data". Prakticky vzato se může jednat o jakýkoliv textový soubor. Je ale nutné ho správně naplnit daty a cestu k němu specifikovat v launch file programu "static_tf_broadcast". Data v textovém souboru mají následující strukturu.

Výpis 2.26: Instalace knihoven

```
(název Tx Ty Tz Rx Ry Rz)                                    1
0 3.15 2.22 1.48 1.57 0 3.14                                 2
```

Následně je potřeba zapnout usb_cam (je nutné specifikovat cestu ke kalibračnímu souboru), marker_detecotr (aruco_ros), locator (aruco_ros_locator) následně spustit "tf_static_broadcast"(aruco_ros_locator).

Závěr

Tato práce se zabývá řešením problematiky lokalizace robota v trojrozměrném prostoru a vývoji programu, který stanoví jeho pozici s aplikací na reálném zařízení. K vývoji v rámci této práce byla využita platforma ROS. Práce byla rozvržena na dvě poloviny.

Čtenáři v těchto částech tak byla nabídnuta přehledná a srozumitelná rešerše, která je nutností pro pochopení problémů a řešení, jež byly použity. První část byla věnována seznámení se s problematikou transformací, jejich popisu a jiných praktických poznatků, které bylo potřeba znát před započetím jakýchkoliv praktických prací. Též bylo nutné se seznámit právě s již zmiňovaným prostředím ROS a knihovnou aruco_ros, která tvořila stěžejní část této práce. Tato část pak byla zakončena vytvořením programu, kterým byla lokalizována kamera vůči pozici značky, fungujícím v simulovaném prostředí. Nutno podotknout, že byl tento program vytvořen čistě za účelem vzdělávání jak mně, tak i čtenářům potenciálně pracujících na shodné problematice. Program byl z velké části převzat z knihovny aruco_ros, který vznikl menšími úpravami mou osobou. Žádné z těchto kódů tak nebyly použity ve výsledné práci. Bylo ale pochopeno, jakým způsobem je kód psán a byly prostudovány funkce, jež tvořily základy pro vlastní tvorbu.

Druhá část práce pak byla zaměřena na praktickou činnost. Po důsledném zvážení všech kladů a záporů, bylo rozhodnuto, že program bude vytvořen v programovacím jazyce Python. Výsledná knihovna obsahuje tři programy, jež každý z nich plní nedílnou část práce za účelem funkční detekce a lokalizace. Veškeré programy byly koncipovány tak, aby byla uživateli nabídnuta co nejvyšší volnost při aplikaci s co nejmenšími nutnými intervencemi. Zároveň byl také kladen velký důraz na automatizaci procesů a tím pádem na celkové pohodlí uživatele. Příkladem programu, který ke zmíněnému pohodlí přispívá, je program, vypočítávající statistické veličiny při měření přesnosti lokalizace. Právě měření přesnosti byla kapitola, která byla zdlouhavě a opatrně vypracována, jelikož se jednalo o důležitou část celkového vývoje. Bylo nutné znát s jakou přesností může být při detekci počítáno. Taktéž bylo nutné znát hodnoty kovarianční matice, která byla využita pro fúzi výsledných dat. Přesnost ale nemohla být zjištěna pouze z provedeného pokusu, a to z důvodu možnosti určení přesnosti v jediné ose. Takto zkonstruovaný experiment tak sloužil pro ověření hodnot z dat, jež byly získány z vědeckého článku zabývajícím se touto problematikou. Bezmála nejdůležitějším programem byla pak samotná detekce. Velký důraz byl opět kladen na automatizaci procesů, ale především na celkovou rychlost a funkčnost. Ne všechny nápady tak byly nakonec aplikovány. Ve výsledku vznikl robustní program, který obsahuje dvě možnosti detekce značek. Zmíněnými módy je možné zvolit, zda budou značky průměrovány, či se bude program orientovat dle

nejbližší z nich. Každý mód se ukázal být výhodný pro jiné účely. Zatímco průměrná hodnota zvyšovala přesnost, nejbližší značka naopak snižovala složitost funkcí. Na konci práce byl program napojen na již existující lokalizační systém na reálném robotu a to pomocí Kalmanova filtru. Hodnoty tak byly zfúzovány. Výsledné měření se prokázalo býti přesnější než se očekávalo a já doufám, že tato práce bude sloužit minimálně jako základ lidem, kteří se v budoucnu budou potýkat s problémy lokalizace.

Literatura

[1] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). *Robot Operating System 2: Design, architecture, and uses in the wild*. Science Robotics, 7. https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[2] Stanford Artificial Intelligence Laboratory et al. (2017). *ROS2 Documentation*. [Online]. California. December https://docs.ros.org/en/humble/index.html.

[3] PenguinMaths. (2020, July 30). *How quaternions produce 3D rotation*. [Video]. YouTube. https://www.youtube.com/watch?v=jTgdKoQv738.

[4] Numberphile. (2016, January 18). *Fantastic Quaternions - Numberphile*. [Video]. YouTube. https://www.youtube.com/watch?v=3BR8tK-LuB0&t=6s.

[5] 3Blue1Brown. (2018, September 6). *Visualizing quaternions (4d numbers) with stereographic projection*. [Video]. YouTube. https://www.youtube.com/watch?v=d4EgbgTm0Bg.

[6] Wolfram Research, Inc. (n.d.). *Rotation Quaternions, and How to Use Them*. https://danceswithcode.net/engineeringnotes/quaternions/quaternions.html.

[7] D. Rose. (2015, May). *Quaternion – from Wolfram MathWorld*. https://mathworld.wolfram.com/Quaternion.html.

[8] D. Rose. (2015, May). *Quaternion – from Wolfram MathWorld*. https://mathworld.wolfram.com/Quaternion.html.

[9] Allversity. (2012, May 30). *How does a camera work?*. [Video]. YouTube. https://www.youtube.com/watch?v=qS1FmgPVLqwl.

[10] Secutek. (n.d.). Ohnisková vzdálenost/úhel záběru objektivu *How does a camera work? (Photography basics explained)*.. Secutek. https://secutek.cz/content/13-ohniskova-vzdalenostuhel-zaberu-objektivu.

[11] Harris, T. (2023, March 8). *How cameras work*. HowStuffWorks. https://electronics.howstuffworks.com/camera.htm.

[12] W-Technika. (n.d.). *CCD vs. CMOS - srovnání senzorů*. W-Technika. https://www.w-technika.cz/ccd-vs-cmos-srovnani-senzoru/1.

[13] Hossein Ashtari. (2023, October 26) *CCD vs. CMOS: 5 Differences To Know*. Spiceworks. https://www.spiceworks.com/tech/tech-general/articles/ccd-vs-cmos/1.

[14] *CCD vs CMOS: A Review of Sensor Technology*. CMOS sensor inc. https://www.csensor.com/ccd-vs-cmos.

[15] MATLAB & Simulink. (n.d.). *What is camera calibration?*. MathWorks. https://www.mathworks.com/help/vision/ug/camera-calibration.html.

[16] Bhatt, D. (2023, June 13). *What is Camera Calibration in Computer Vision?*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-for-camera-calibration-in-computer-vision/.

[17] F. J.Romero-Ramirez, R. Muñoz-Salinas, R. Medina-Carnicer. (2018). *Speeded up detection of squared fiducial markers*. Image and Vision Computing, vol 76. pages 38-47. https://www.researchgate.net/publication/325787310_Speeded_Up_Detection_of_Squared_Fiducial_Markers.

[18] S. Garrido-Jurado, R. Muñoz Salinas, F.J. Madrid-Cuevas, R. Medina-Carnicer. (2016). *Generation of fiducial marker dictionaries using mixed integer linear programming*. Pattern Recognition:51, pages 481-491. https://www.researchgate.net/publication/282426080_Generation_of_fiducial_marker_dictionaries_using_Mixed_Integer_Linear_Programming.

[19] Garrido-Jurado, S., Muñoz-Salinas, R., Madrid-Cuevas, F. J., & Marín-Jiménez, M. J. (2014). *Automatic generation and detection of highly reliable fiducial markers under occlusion*. Pattern Recognition, 47(6), 2280–2292. https://doi.org/10.1016/j.patcog.2014.01.005.

[20] Lee, S. (2022, February 5). *Understanding homography (a.k.a perspective transformation)*. Medium. https://towardsdatascience.com/understanding-homography-a-k-a-perspective-transformation-cacaed5ca17.

[21] Science, B. O. C., & Science, B. O. C. (2023, May 6). *Understanding OTSU's method for image segmentation | Baeldung on Computer Science*. Baeldung on Computer Science. https://www.baeldung.com/cs/otsu-segmentation.

[22] *ArUco: An efficient library for detection of planar markers and camera pose estimation*. Applicaciones de la Visión Artifical https://www.uco.es/investiga/grupos/ava/portfolio/aruco/.

[23] R. Fisher, S. Perkins, A. Walker and E. Wolfart. (2003) *Canny Edge detector*. https://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm.

[24] *Standard Deviation*. National Library of Medicine https://www.nlm.nih.gov/oet/ed/stats/02-900.html.

[25] Jim Frost *Root Mean Square Error (RMSE).* Statistics By Jim https://statisticsbyjim.com/regression/root-mean-square-error-rmse/.

[26] Satish Gunjal (2021) *Root Mean Square Error (RMSE).* Kaggle https://www.kaggle.com/discussions/general/215997.

[27] *Covariance Matrix).* Cuemath https://www.cuemath.com/algebra/covariance-matrix/.

[28] Greg Welch and Gary Bishop (Sep. 17, 1997) *An Introduction to the Kalman Filter.* https://perso.crans.org/club-krobot/doc/kalman.pdf.

[29] Kateřina Frončková, Pavel Pražák (Jan. 22, 2019) *Kalman Filter and Time Series.* University of Hradec Kralove https://www.cuemath.com/algebra/covariance-matrix/.

[30] Dan Simon (June 2001) *Covariance Matrix).* Embedded Systems Programming https://abel.math.harvard.edu/archive/116_fall_03/handouts/kalman.pdf.

[31] *What is the Kalman Filter?).* Iain Explains Signals, Systems, and Digital Comms https://www.youtube.com/watch?v=0iUS2926nQM&t=183s.

[32] Gensler, J. (2021, December 26). *ROS2 Image Pipeline Tutorial.* Medium. https://jeffzzq.medium.com/ros2-image-pipeline-tutorial-3b18903e7329.

[33] Wikipedia contributors. (2023, November 6). *Lennal.* Wikipedia. https://en.wikipedia.org/wiki/Lenna.

[34] *Python vs. C++: Key differences and uses.* Ionos. https://www.ionos.com/digitalguide/websites/web-development/python-vs-c/.

[35] *Differences b/w C++ and Python.* Javatpoint. https://www.javatpoint.com/cpp-vs-python.

[36] *Python vs. C++: Key differences and uses.* Ionos. https://www.ionos.com/digitalguide/websites/web-development/python-vs-c/.

[37] *What is the Python Interpreter? (How does Python Work?).* Asternerd. https://www.youtube.com/watch?v=BkHdmAhapws.

[38] *Python Vs C++: Overview, Similarities & Key Differences).* Great Learning. https://www.mygreatlearning.com/blog/python-vs-cpp/.

[39] *What Is Asynchronous Programming? (And When To Use It).* Indeed. https://www.indeed.com/career-advice/career-development/asynchronous-programming.

[40] Scharon Harding (June 17, 2022) *What Is a CPU Core? A Basic Definition.* Tom's Hardware. https://www.tomshardware.com/news/cpu-core-definition,37658.html.

[41] Song Ho Ahn (2021) *Quaternion to Rotation Matrix.* Songho. https://www.songho.ca/opengl/gl_quaternion.html.

[42] *Rotation Matrix.* Cuemath. https://www.cuemath.com/algebra/rotation-matrix/.

[43] M. Kalaitzakis, S. Carroll, A. Ambrosi, C. Whitehead, N. Vitzilaios *Experimental Comparison of Fiducial Markers for Pose Estimation.* IEEE. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9213977.