Universitat de Girona
Escola Politècnica Superior

Projecte fi de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Sistema de localització pel robot Turlelbot 2 basat en un algoritme EKF-SLAM

Document: Memòria

Alumne: Pau López Ramió

Tutor: Marc Carreras Pérez i Narcís Palomeras Rovira
Departament: Arquitectura i Tecnologia de Computadors
Àrea: Arquitectura i Tecnologia de Computadors

Convocatòria (mes/any): Setembre 2021

Projecte Fi de Grau

Sistema de localització pel robot Turlelbot 2 basat en un algoritme EKF-SLAM

Autor:
Pau López Ramió

Setembre 2021

Grau en Enginyeria Informàtica

Tutors:
Marc Carreras Pérez
Narcís Palomeras Rovira

Índex

1 Introducció **1**
1.1 Part complementària 1
1.2 Part diferenciada corresponent a aquest TFG 2

2 Estudi de viabilitat **3**

3 Metodologia **4**

4 Planificació **5**

5 Marc de treball i conceptes previs **6**
5.1 Marc de treball 6
5.2 Conceptes previs 6
5.2.1 ROS (Robot Operating System) 6
5.2.2 ArUco 7
5.2.3 SLAM (Simultaneous localization and mapping) 7
5.2.4 EKF (Extended Kalman filter) 8

6 Requisits del sistema **9**

7 Estudis i decisions **10**
7.1 Llenguatge de programació 10
7.1.1 C++ 10
7.1.2 Python 11
7.1.3 Llenguatge escollit 11
7.2 Programari utilitzat 12
7.2.1 Bitbucket 12
7.2.2 Overleaf 12
7.2.3 Jupyter notebook 13
7.2.4 Draw.io 13
7.2.5 Google Meet 13
7.2.6 Notepad++ 14
7.2.7 Editor de text Ubuntu 20.04 14
7.3 Detector de markers 15

8 Anàlisi i disseny del sistema **16**
8.1 Connexió Wi-Fi als robots Turtlebot 2 16
8.1.1 Accessoris addicionals 17
8.2 Ús d'un nou sensor, càmera Intel RealSense 17
8.3 Paquet ArUco 18
8.3.1 Marker Array 18
8.3.1.1 Marker 19
8.3.1.1.1 Header 20
8.3.1.1.2 Pose 20
8.3.1.1.2.1 Point 20
8.4 Paquet ar_marker_corrector 21
8.4.1 PointStamped 21
8.4.2 BetterPoint 22
8.4.3 PointArray 22
8.5 EKF-SLAM 23
8.5.1 ExtendedKalmanFilter 23
8.5.1.1 prediction 23
8.5.1.2 update 23
8.5.2 TurtlebotVelocityEkf 25
8.5.2.1 calculate_f 25
8.5.2.2 calculate_A 25
8.5.2.3 calculate_W 25
8.5.2.4 calculate_h 25
8.5.2.5 calculate_H 26
8.5.3 VelocityIntegrator 26
8.5.3.1 calculate_f 26
8.5.3.2 calculate_A 26
8.5.3.3 calculate_B 27
8.5.3.4 calculate_W 27
8.5.4 TurtlebotEkfSlam 27
8.5.4.1 compatibilityTest 28
8.5.4.2 calculate_f 28
8.5.4.3 calculate_A 28
8.5.4.4 calculate_W 28
8.5.4.5 calculate_h 28
8.5.4.6 calculate_H 28
8.5.4.7 state_augmentation 29

9 Implementació i proves **30**
9.1 Integració de la càmera Intel RealSense D435i 30
9.1.1 Instal·lació en Ubuntu 20.04 30

9.1.2 Calibratge . . . 31
9.1.2.1 Instal·lació de les dependències de 3rs . . . 31
9.1.2.2 Intal·lació del paquet mitjançant Amazon AWS . 31
9.1.2.3 Execució de l'eina de calibratge . . . 32
9.1.3 Instal·lació en ROS Noetic . . . 32
9.2 Integrar un nou punt d'accés als robots Turtlebot 2 . . . 33
9.2.1 Intal·lació del router . . . 33
9.2.2 Manual de connexió . . . 34
9.3 Integració del paquet aruco_ros . . . 35
9.4 Error amb el càlcul de distàncies . . . 36
9.5 Obtenció d'un dataset . . . 38
9.5.1 Construcció d'un entorn real . . . 38
9.5.1.1 Col·locació de les ArUco . . . 39
9.5.2 Construcció d'un entorn simulat . . . 41
9.5.2.1 Col·locació de les ArUco . . . 42
9.5.3 Enregistrar el recorregut per l'entorn . . . 44
9.6 Creació de ar_marker_corrector . . . 45
9.6.1 Crear el paquet de ROS . . . 45
9.6.2 Definir els missatges a publicar (msg) . . . 45
9.6.3 Crear el programa que corregeixi i publiqui les dades (src) . 46
9.6.3.1 Re-calibratge de la càmera . . . 46
9.6.3.1.1 Obtenció dels punts . . . 47
9.6.3.1.2 Obtenció de totes les dades . . . 48
9.6.3.1.3 Creació de un pla polinòmic . . . 49
9.6.4 Crear el *laucher* del paquet (launch) . . . 50
9.7 Estimació de la incertesa . . . 50
9.7.1 Obtenció de totes les dades . . . 50
9.7.2 Creació d'una corba polinòmica . . . 51
9.8 Implementació de les equacions EKF-SLAM . . . 52

10 Implantació i resultats **53**
10.1 Implantació . . . 53
10.2 Resultats . . . 54
10.2.1 Robot muntat i totalment equipat . . . 54
10.2.2 EKF-SLAM . . . 55
10.2.2.1 Entorn real . . . 56
10.2.2.2 Entorn simulat . . . 58
10.2.3 Pràctica de robòtica mòbil . . . 59

11 Conclusions **60**

12 Treball futur **61**

13 Annexos **62**

13.1 Script mesurador de distàncies ArUco 62
13.2 Paquet ar_marker_corrector 64
13.2.1 Corrector de markers 64
13.2.2 Generador de soroll per l'odometria 67
13.3 EKF-SLAM 71
13.4 Generador de models ArUco per gazebo 77
13.5 Pràctica de robòtica mòbil 81

Bibliografia **87**

Capítol 1

Introducció

El grup de visió per computador i robòtica vol actualitzar les plataformes Turtlebot 2 disponibles al laboratori de robòtica de l'EPS a nivell de hardware i software. Per això s'han definit dos treballs de final de grau (TFG), aquest i el TFG de l'estudiant Teng Liu Chen (u1940482), que, tot i compartir alguns punts bàsics, estan clarament diferenciats. En ambdós projectes l'objectiu final és el de desenvolupar un sistema de localització en temps real, utilitzant el middleware Robot Operating System (ROS), però fent ús d'algorismes i sensors completament diferents. Així doncs, l'abast del projecte es divideix amb la part compartida i la individual.

1.1 Part complementària

- Actualitzar tant el sistema operatiu com la versió de ROS dels robots Turtlebot 2 a l'última versió. Aquesta part serà desenvolupada pels dos estudiants per tal d'adquirir coneixements de Linux/ROS. Es realitzarà un document tècnic descrivint els passos seguits que no formarà part del cos de cap de les dues memòries de TFG.

- Desenvolupar un entorn de simulació utilitzant el software de codi obert Gazebo en que es simuli els robots Turtlebot 2, equipats amb els nous sensors, així com reproduir de forma virtual l'entorn del laboratori de robòtica de l'EPS. Aquesta tasca serà liderada i presentada per en Teng.

- Integrar un nou punt d'accés als robots Turtlebot 2 per tal de que varis usuaris puguin treballar amb ells sense interferir-se els uns amb els altres i desenvolupar un manual de connexió pels alumnes. Aquesta part serà liderada i presentada per mi.

1.2 Part diferenciada corresponent a aquest TFG

- Integrar una càmera Intel RealSense al robot i calibrar-la.
- Mesurar la incertesa que s'obté quan s'estima la posició relativa a una marca de realitat augmentada utilitzant la llibreria ROS aruco_ros i la càmera RealSense.
- Desenvolupar un sistema de localització i mapeig simultanis (SLAM) utilitzant l'algorisme Extended Kalman Filter capaç d'integrar la odometria del robot Turtlebot 2 i la posició relativa a un conjunt de marques de realitat augmentada vistes des de la càmera Intel RealSense. El qual s'utilitzara com a practica per a futurs alumnes de robotica

Capítol 2
Estudi de viabilitat

Per desenvolupar aquest projecte, els requeriments de hardware són: un ordinador (en el nostre cas un portàtil Lenovo 520S), el Turtlebot 2, una càmera Intel RealSense, un router i un adaptador type-c a ethernet; aquests han estat proporcionats per la UdG. En quant al software, treballarem sobre Ubuntu 20.04 i ROS noetic, per tant, no hi ha cap cost afegit respecte el software. En quant al cost total del projecte, només s'ha de comptar amb la mà d'obra, el router i l'adaptador, ja que la UdG ja disposava prèviament de la resta del hardware i tot el software utilitzat és lliure. Tot i això, el cost aproximat del portàtil són 800€, el Turtlebot 2 uns 700€, la càmera uns 250€, el router uns 30€ i l'adaptador uns 10€. Fent un total aproximat de 1790€.

Pel que fa a la utilització d'un algoritme EKF-SLAM com a mètode de localització, juntament amb *markers* ArUco com a *landmarks* per un Tutlebot 2, és necessari saber si es podrà desenvolupar en el termini d'un semestre. Afortunadament, es tracta d'un algoritme àmpliament utilitzat i també treballat a classe, el qual facilita la implementació de variacions. A més, els *markers* ArUco també són utilitzats en molts projectes d'investigació relacionats amb la realitat augmentada i consten d'un gran suport per part dels seus desenvolupadors —els professors del grup AVA de la Universidad de Córdoba—, que ens segueixen proveint d'actualitzacions i nous projectes. [ArU 021] [Kalman 1960]

Amb tot això explicat prèviament, es pot seguir endavant amb el projecte sabent que el nostre projecte és totalment viable i només depèn del nostre esforç per poder assolir els requeriments desitjats.

Capítol 3

Metodologia

Per realitzar el projecte de la manera més eficient possible i escollint d'entre els mètodes apresos en els estudis, es va decidir el mètode Scrum com al més idoni per aquesta tasca.

Scrum és un marc de treball que utilitza les metodologies àgils per tal de desenvolupar, entregar i mantenir productes en un entorn complex amb especial èmfasi en el desenvolupament de software. Està dissenyat per equips de 10 membres o menys, que desglossen les feines en objectius que poden ser completats en terminis de temps determinats anomenats *sprints*, que no poden ser més llargs d'un mes i, habitualment, de dues setmanes. L'equip Scrum comenta el progrés en petites reunions de menys de 15 minuts cada dia, anomenats "Scrums diaris". Al final de cada *sprint*, l'equip té dos reunions: la revisió de l'*sprint* on es mostra la feina feta, i la retrospectiva de l'*sprint* on l'equip reflexiona i proposa millores.

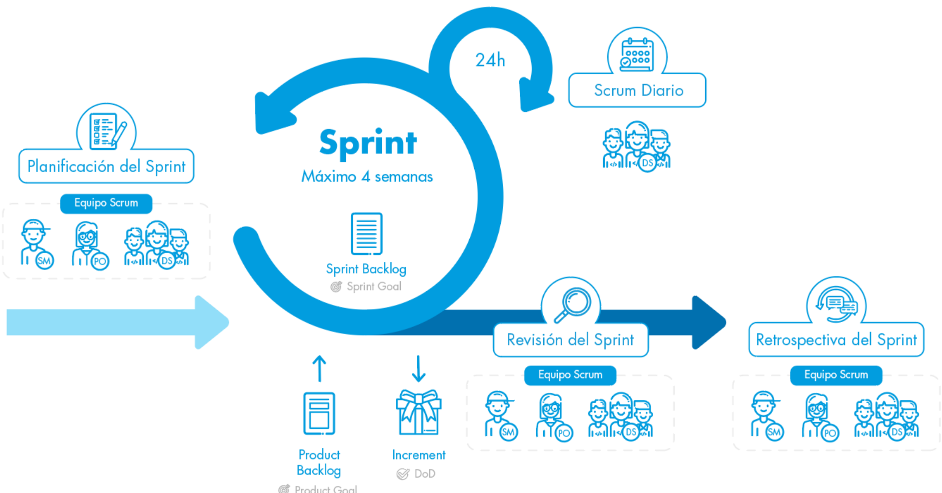


Figura 3.1: Diagrama de flux del mètode Scrum

Capítol 4
Planificació

Tal i com s'explica en el capítol anterior, utilitzem el mètode Scrum, en el qual ens marquem objectius a assolir cada 2 setmanes i després ens reunim per comentar els resultats. En el següent diagrama es poden observar millor les diferents tasques a realitzar i el temps en les quals s'han dut a terme.

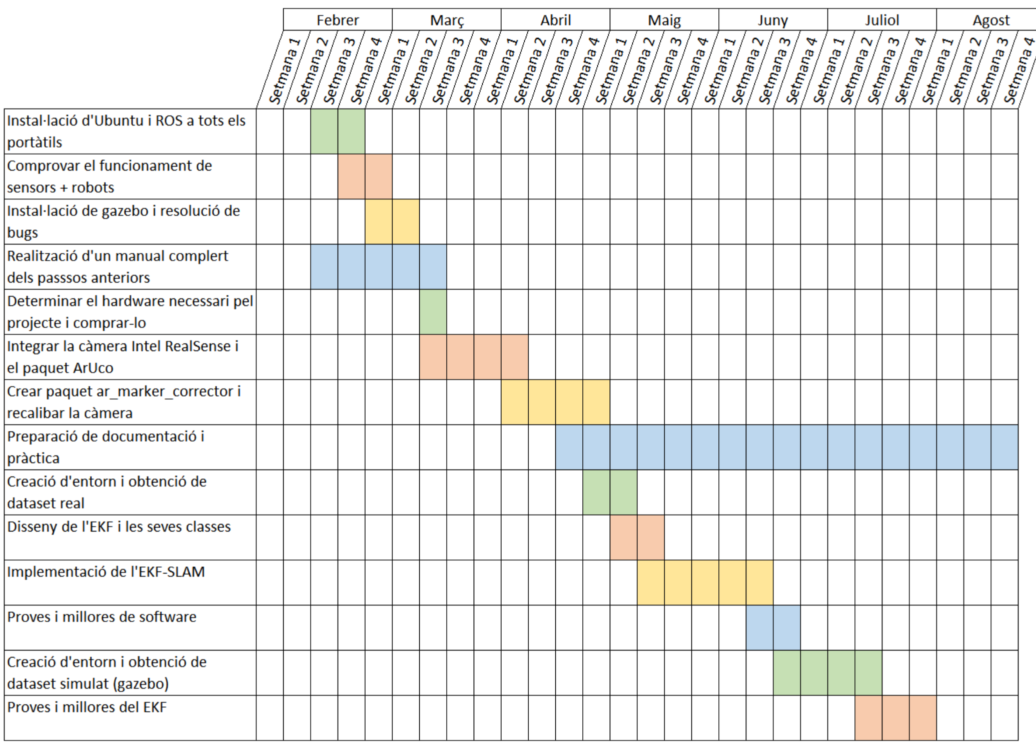


Figura 4.1: Diagrama de Gantt amb la planificació del projecte

La durada del treball ha estat de aproximadament 26 setmanes de les quals 20 es realitzaven reunions amb el tutor periòdicament cada 1-2 setmanes. Aquestes reunions duraven entre 1,5 i 2h el que fa un total aproximat de 30-40h. Pel que fa al treball individual es molt complicat comptabilitzar les hores. Tot i aixi aproximadament dedicava unes 4h al dia durant mínim 4 dies de la setmana per tant he dedicat unes 420-450h en la realització d'aquest projecte.

Capítol 5

Marc de treball i conceptes previs

Per tal d'entendre millor aquest treball és necessària l'explicació dels següents temes clau.

5.1 Marc de treball

Com s'ha comentat a la introducció, aquest projecte hauria de servir com a pràctica per a futurs alumnes de la UdG, per tant, s'ha fet amb col·laboració amb el departament d'Arquitectura i Tecnologia de Computadors de la UdG i el CIRS. Especialment, amb en Narcís Coll i Marc Carreras com a tutors del projecte, així com la resta d'integrants del CIRS que em van resoldre molts dubtes de ROS, EKF i hardware; com la creació d'un suport que muntés la càmera Intel RealSense al Turtlebot 2.

5.2 Conceptes previs

En aquest apartat es descriuran els conceptes base que aniran apareixent al llarg d'aquest document per tal de poder fer un correcte seguiment del projecte.

5.2.1 ROS (Robot Operating System)

El ROS és un *middleware* que tot i anomenar-se *Operating System*, no es tracta d'un sistema operatiu sinó, més aviat, es tracta d'una col·lecció de *frameworks* per al desenvolupament de software robòtic. Bàsicament, és una eina que ens permet unificar la comunicació entre els actuadors, sensors i controladors del robot de manera més simple.

Figura 5.1: Dos nodes de ROS

5.2.2 ArUco

És una llibreria per a la realització d'aplicacions de realitat augmentada que es basa en la detecció d'uns *markers* (semblants als codis QR) amb els quals som capaços de calcular la posició relativa a nosaltres i la seva orientació. Aquesta llibreria ha estat implementada per l'equip d'investigació d'aplicacions de visió artificial (A.V.A) de la Universidad de Córdoba. [ArU 021]

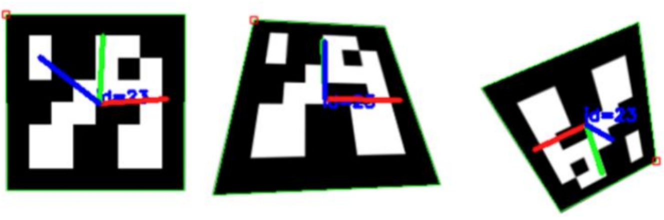


Figura 5.2: Markers Aruco

5.2.3 SLAM (Simultaneous localization and mapping)

Com bé diu el seu nom, es tracta de construir i/o actualitzar un mapa en un entorn desconegut i, simultàniament, mantenir la localització d'un mateix dins d'aquest. Ser capaç de solucionar aquest problema pot semblar impossible, però hi han diversos algoritmes que l'han solucionat. D'entre ells es troba el **filtre extes de Kalman (EKF)** el qual utilitzarem en aquest projecte.

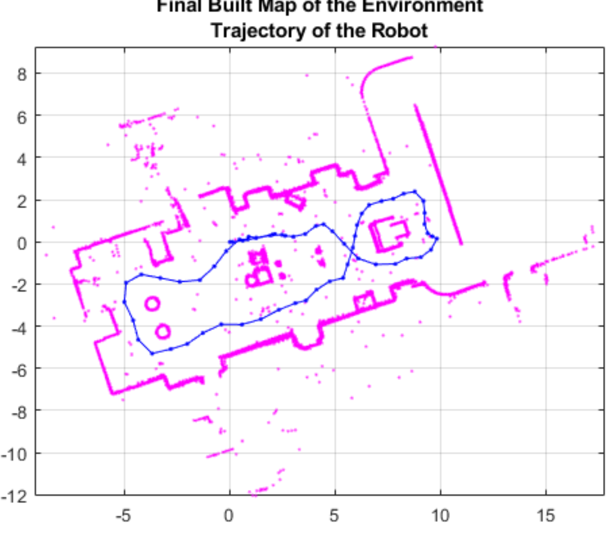


Figura 5.3: Mapa i trajectòria SLAM

5.2.4 EKF (Extended Kalman filter)

És un algoritme que utilitza una serie de mesures lligades a un soroll i una incertesa per estimar de manera òptima les variables que ens interessen, en el nostre cas, la posició i orientació del robot. Ja que el nostre sistema no és lineal, cal utilitzar la versió *extended* del mateix. Degut a les seves grans prestacions, és considerant un estàndard en la teoria d'estimació d'estats no lineals en la gran majoria de sistemes de navegació i GPS. [Kalman 1960] [Courses 2006]

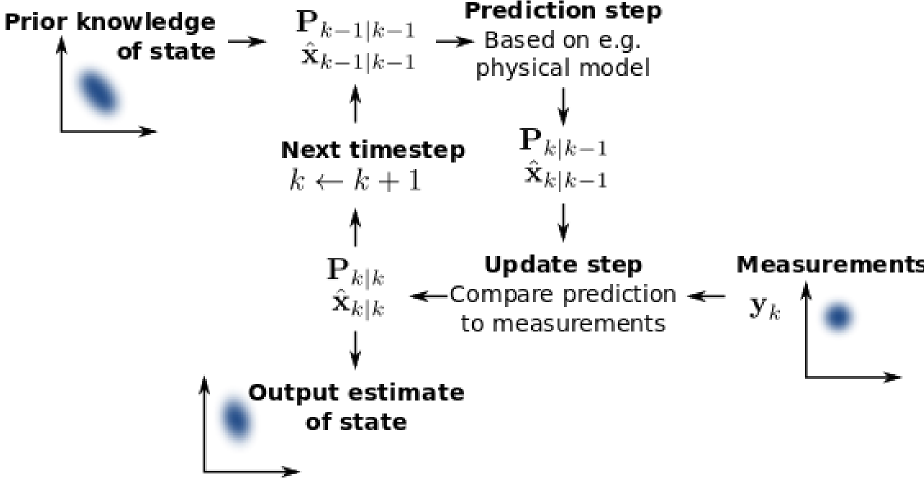


Figura 5.4: Estructura bàsica del filtre de Kalman

Com es pot observar en la imatge el funcionament bàsic és el següent:

- Pas 1: Rebem un estat amb incertesa.
- Pas 2: Utilitzant l'estat rebut i l'anterior, fem una predicció.
- Pas 3: Actualitzem el nostre estat amb les mesures rebudes del exterior.
- Pas 4: Publiquem l'estat i la retornem al Pas 2.

Capítol 6
Requisits del sistema

Per tal que els alumnes puguin entendre i realitzar la pràctica és necessari complir uns requisits dividits en funcionals i no funcionals.

Requisits funcionals:

- El sistema reconeix tots els actuadors i sensors
- El sistema reconeix els *markers* Aruco
- El sistema calibra els sensors correctament
- El sistema és capaç de localitzar-se correctament
- L'estudiant disposa de l'estructura bàsica per construir el seu EKF
- L'estudiant pot connectar-se al robot via Wi-Fi i controlar-lo

Requisits no funcionals:

- El sistema utilitza una càmera Intel RealSense
- El sistema utilitza *markers* ArUco com a *landmarks*
- El sistema utilitza l'algoritme EKF per localitzar-se

Els requisits tècnics per part de l'estudiant són molt simples, per poder realitzar la pràctica necessita un PC capaç d'executar Python o fitxers Jupyter. En quant al control remot del robot és necessari un PC amb Ubuntu i ROS noetic. En cap dels dos casos és necessari un hardware avançat.

Capítol 7

Estudis i decisions

7.1 Llenguatge de programació

Per a realitzar aquest projecte amb ROS, ens veiem limitats a dos llenguatges de programació:

7.1.1 C++

Pros:

- Execució molt ràpida.
- Degut a la necessitat de compilar, permet localitzar errors abans d'executar.
- Disposa d'un gran abast de llibreries amb infinites possibilitats.
- És un estàndard en la indústria robòtica i sol ser un requisit primordial per treballar-hi.

Cons:

- És un llenguatge complex amb una corba d'aprenentatge molt alta.
- Per crear un projecte petit és necessari molt de codi.
- Pot ser dificil de llegir i entendre.
- El procés de debugar errors pot ser llarg i complex.

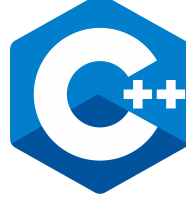


Figura 7.1: Logo de C++

7.1.2 Python

Pros:

- És molt fàcil i ràpid construir un projecte i executar-lo.
- No és necessari compilar.
- Es considera un llenguatge senzill de aprendre.
- Té la capacitat de crear programes complexos amb molt poc codi.
- El codi sol ser bastant llegible i entenedor.
- Disposa d'un gran abast de llibreries amb infinites possibilitats.
- És més fàcil treballar amb certes funcions de ROS ja que s'encarrega de gestionar-les la pròpia API de Python.

Cons:

- S'executa amb més lentitud.
- És més probable que falli durant l'execució.
- A mesura que el projecte creix, és habitual acabar amb un codi poc entenedor i difícil de treballar.
- L'API de ROS en Python és més restrictiva.
- Segons la complexitat del projecte, es pot quedar curt.

Figura 7.2: Logo de Python

7.1.3 Llenguatge escollit

Un cop exposats el pros i cons dels dos llenguatges, es va decidir treballar amb Python, ja que es tracta d'un programa relativament senzill i degut a que és molt probable que els estudiants que realitzin la pràctica tinguin poca experiència en programació.

7.2 Programari utilitzat

Durant la realització d'aquest projecte s'han utilitzat diversos programes dividits en dos entorns: Windows 10 i Ubuntu 20.04. A continuació, s'expliquen quins són els programes i perquè s'han escollit.

7.2.1 Bitbucket

És necessari l'ús d'un repositori de codi, ja sigui per al control de versions o com a backup.Vam escollir Bitbucket ja que permet crear repositoris privats i escollir qui hi pot accedir. Això és important, ja que el codi d'aquest projecte referent al EKF-SLAM no pot ser públic, ja que sinó els alumnes podrien trobar-ne la solució però, a la vegada, és necessari que l'equip de la UdG hi tingui accés.

Figura 7.3: Logo de Bitbucket

7.2.2 Overleaf

Per tal de realitzar tota la documentació s'ha utilitzat LaTeX, ja que ens va ser introduït en els estudis i resulta molt pràctic i elegant, sobretot a l'hora de documentar software. Com a editor de LaTeX es va escollir Overleaf, ja que proporciona una gran versatilitat, degut a que s'utilitza des del navegador i permet editar el document des de qualsevol lloc. També, resulta molt útil per a les revisions ja que permet compartir fàcilment amb el tutor i aquest et pot corregir i comentar aspectes del document.

Figura 7.4: Logo d'Overleaf

7.2.3 Jupyter notebook

Per dur a terme la pràctica de robòtica es va utilitzar el mateix sistema amb el que ens presentaven les pràctiques de robòtica mòbil. Aquest editor de text ens permet integrar text per les explicacions i blocs de codi executable des de la pàgina web. D'aquesta manera, es pot llegir la teoria, implementar i executar el codi allà mateix.

Figura 7.5: Logo de Jupyter notebook

7.2.4 Draw.io

Per realitzar tot tipus de diagrames i esquemes s'ha utilitzat Draw.io, ja que va ser de gran utilitat durant els estudis i facilita la feina de disseny software. És senzill d'utilitzar i permet guardar els projectes al núvol.

Figura 7.6: Logo de Draw.io

7.2.5 Google Meet

Degut a les circumstàncies excepcionals de la pandèmia, no sempre ens va ser possible realitzar reunions de manera presencial. Degut a això, a arreu s'utilitzen diversos serveis de videoconferència. En el nostre cas, vam utilitzar Google Meet, ja que complia els requisits bàsics (vídeo, xat, compartir pantalla, etc.) i, a més, ambdós ja disposàvem de compte de Google (de la UdG).

Figura 7.7: Logo de Google Meet

7.2.6 Notepad++

Per poder editar els diversos documents amb diferents extensions d'aquest projecte, era necessari utilitzar un editor que fos capaç de obrir-los i també proporcionar algun tipus d'ajuda per escriure. Es va escollir Notepad++, ja que en L'entorn Windows és àmpliament utilitzat i permet una gran versatilitat tant amb llenguatges diferents, com amb tipus de fitxers.

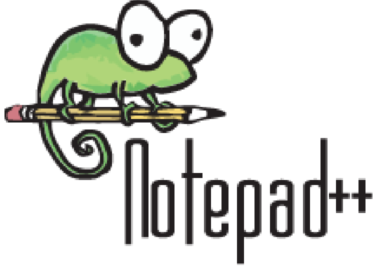


Figura 7.8: Logo del Notepad++

7.2.7 Editor de text Ubuntu 20.04

De la mateixa manera que amb el Notepad++ necessitava una alternativa per Ubuntu, tot i existir programes de tercers com NotepadQQ, no ha estat necessari el seu ús, ja que l'editor de text predeterminat d'Ubuntu ja aporta les funcionalitats bàsiques per poder treballar còmodament.

Figura 7.9: Logo del editor per defecte de Ubuntu

7.3 Detector de markers

Per detectar *markers* necessitem una llibreria capaç de processar l'imatge de la càmera i detectar si apareix algun dels *markers*. Existeixen una gran varietat de llibreries destinades a la detecció de diferents tipus de *markers*. Vam decidir utilitzar la llibreria ArUco, ja que es tracta d'una llibreria àmpliament utilitzada i també per la comoditat de ser una llibreria amb la qual ja he treballat anteriorment durant els estudis.

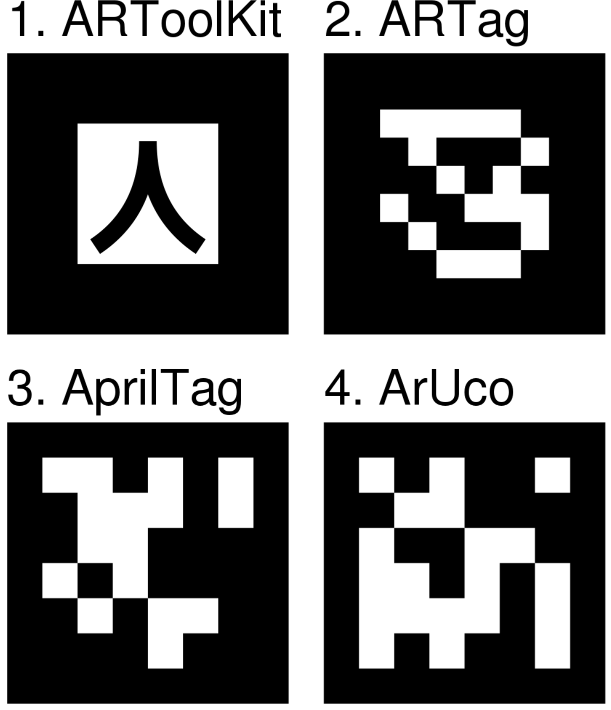


Figura 7.10: Comparativa de diferents tipus de markers

Capítol 8

Anàlisi i disseny del sistema

Per a la completa realització d'aquest treball és necessari complir tots els requisits correctament, per això, cadascun d'ells s'ha estudiat per tal de poder obtenir una solució que els compleixi.

8.1 Connexió Wi-Fi als robots Turtlebot 2

Inicialment, els alumnes es trobaven amb problemes de connexió amb el robot degut a que el portàtil que s'utilitzava no era capaç de suportar un nombre elevat de connexions simultànies. Amb aquest aspecte en ment, es va decidir integrar un router que actués com a *access point* al Turtlebot 2. El nostre objectiu era aconseguir màxima velocitat, mínima latència i múltiples connexions a grans distàncies. D'entre diversos candidats, es va escollir el Xiaomi Mi A4 Gigabit edition, el qual disposa de 3 ports Gbit, 4 antenes Wi-Fi (2X5dbi i 2X6dbi) a 2,4Ghz i 5Ghz.
A més, s'hi afegeix el fet que presenta una mida reduïda ideal per al Turtlebot 2.

Figura 8.1: Router Xiaomi Mi A4

8.1.1 Accessoris addicionals

Els nous portàtils que aniran connectats al Turtlebot 2 no disposen de connexió ethernet, la qual cosa ens obliga a afegir una adaptador addicional, concretament un connector USB type-c a ethernet Gbit.

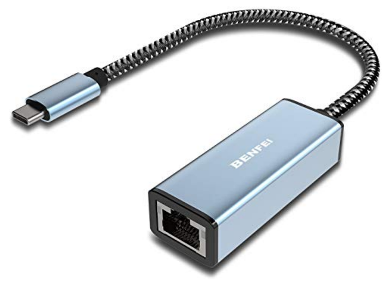
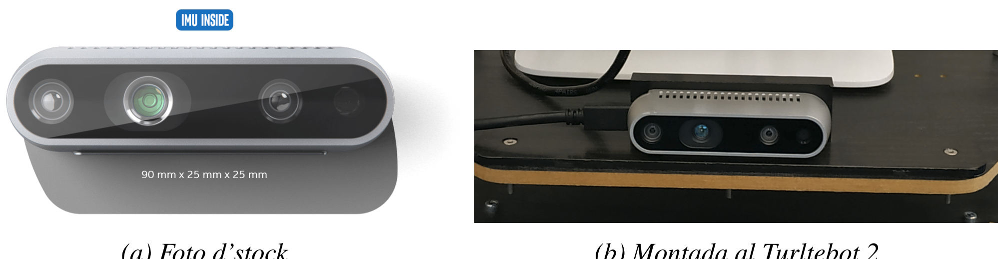


Figura 8.2: Adaptador USB type-c a ethernet

8.2 Ús d'un nou sensor, càmera Intel RealSense

Com s'ha comentat anteriorment, ens interessa renovar la càmera que munta el Turtltebot 2, ja que es tracta d'una Kinect del 2010 que ha quedat bastant antiquada. La solució es tracta d'una càmera Intel RealSense D435i, que no només té més resolució i millor definició de profunditat, sinó que també és molt més compacte i eficient, ja que només necessita 5V, 1A comparats amb els 12V de la Kinect. Aquesta càmera consta de dos sensors estereoscòpics amb una resolució de 720p a 90fps i un sensor RGB amb una resolució de 1080p a 30fps. [Int 021d] [Int 021b]



(a) Foto d'stock *(b) Montada al Turltebot 2*

Figura 8.3: Càmera Intel RealSense

En quant al software serà necessari instal·lar tots els *drivers* i paquets ROS necessaris per a poder utilitzar-la.

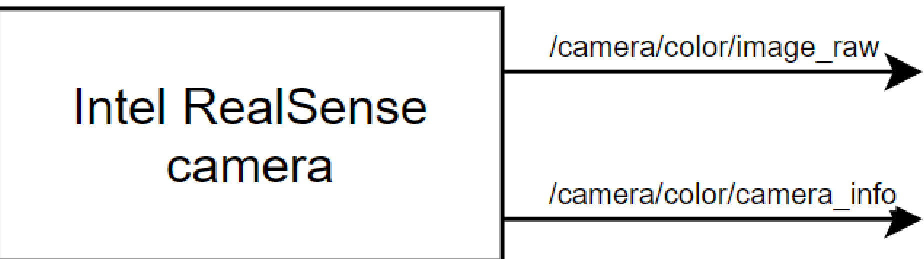


Figura 8.4: Topics publicats per la càmera

Per continuar amb el següent requeriment, caldrà subscriure's als tòpics mostrats a la imatge.

8.3 Paquet ArUco

Un cop definits els tòpics de la càmera, tal i com s'observa a la Figura 8.4, es necessitarà un paquet que s'encarregui de processar la informació proporcionada per la càmera i ens retorni els *markers* trobats a la imatge.

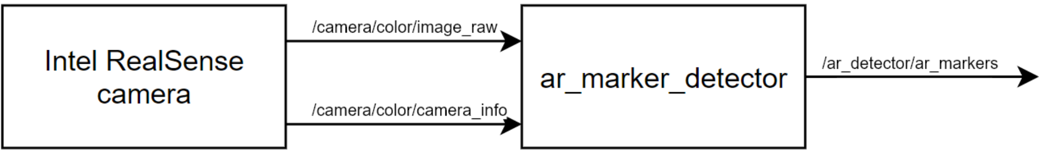


Figura 8.5: Tòpics publicats pel paquet ArUco

8.3.1 Marker Array

Aquest paquet ens publica un missatge estandar de ROS anomenat *visualization_msgs/MarkerArray.msg*, que com bé indica el seu nom, conté una *array* de *markers* [Mar 2021b]:

```
Marker[] markers
```

8.3.1.1 Marker

Aquest tipus esta definit pel missatge *visualization_msgs/Marker.msg*, el qual conté el següent [Mar 2021a]:

```
uint8 ARROW=0
uint8 CUBE=1
uint8 SPHERE=2
uint8 CYLINDER=3
uint8 LINE_STRIP=4
uint8 LINE_LIST=5
uint8 CUBE_LIST=6
uint8 SPHERE_LIST=7
uint8 POINTS=8
uint8 TEXT_VIEW_FACING=9
uint8 MESH_RESOURCE=10
uint8 TRIANGLE_LIST=11
uint8 ADD=0
uint8 MODIFY=0
uint8 DELETE=2
uint8 DELETEALL=3

std_msgs/Header header
string ns
int32 id
int32 type
int32 action
geometry_msgs/Pose pose
geometry_msgs/Vector3 scale
std_msgs/ColorRGBA color
duration lifetime
bool frame_locked
geometry_msgs/Point[] points
std_msgs/ColorRGBA[] colors
string text
string mesh_resource
bool mesh_use_embedded_materials
```

Aquest missatge ens proporciona molta informació, però per dur a terme el nostre propòsit només en necessitem dos: *Header header* i *Pose pose*.

8.3.1.1.1 Header

El *header* conté informació primordial per a la temporització i origen de les dades. Els paràmetres estan definits pel missatge *std_msgs/Header.msg*, el qual conté el següent [Hea 2020]:

```
uint32 seq
time stamp
string frame_id
```

8.3.1.1.2 Pose

Com bé indica el seu nom, ens indica la posició i orientació del *marker* en referència al *frame* definit dins del *header*. Els paràmetres estan definits pel missatge *geometry_msgs/Pose.msg*, el qual conté el següent [Pos 2021]:

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

Ja que, en el nostre cas, només ens interessa la posició del *marker*, només utilitzarem *Point position*.

8.3.1.1.2.1 Point

Com bé indica el seu nom, ens indica la posició del *marker* en referència al *frame* definit dins del *header*. Els paràmetres estan definits pel missatge *geometry_msgs/Point.msg*, el qual conté el següent [Poi 2021a]:

```
float64 x
float64 y
float64 z
```

8.4 Paquet ar_marker_corrector

El nostre objectiu actual és crear un paquet que sigui capaç de llegir la informació del paquet ArUco i proporcionar-nos les posicions dels *markers* respecte el robot.

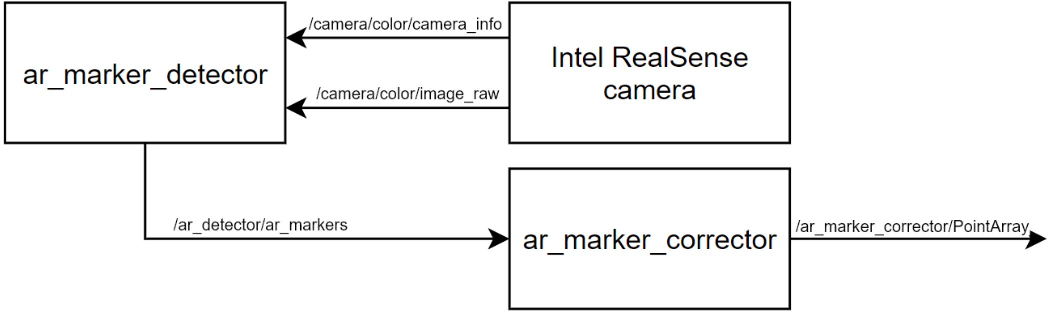


Figura 8.6: Tòpics publicats pel paquet ar_marker_corrector

Per aconseguir aquest objectiu, en primer lloc, hem de pensar en el missatge més bàsic que necessitem. Ja que allò que volem és conèixer la posició del *marker* respecte al robot, primer necessitarem un *Point*.

Com s'ha vist a l'apartat 8.3.1.1.2.1, un *Point* està format per [Poi 2021a]:

```
float64 x
float64 y
float64 z
```

Tot i que en el nostre cas només necessitem 2 dimensions, l'utilitzarem igualment, ja que en un futur podrien interessar utilitzar les 3 dimensions.
Així doncs, ja tenim el missatge més bàsic, però no ens proporciona informació suficient. No obstant, buscant dins dels missatges estàndard de ROS vaig trobar un tipus que ens seria molt útil per representar la posició i el seu context.

8.4.1 PointStamped

Aquest tipus ens permet emmagatzemar un punt definit en l'espai i dotar-lo de context, ja que incorpora el tipus *Point* i el tipus *header* [Poi 2021b].

```
std_msgs/Header header
geometry_msgs/Point point
```

Arribats a aquest punt, hem esgotat els tipus estàndard i a partir d'ara haurem de crear missatges propis.

8.4.2 BetterPoint

L'objectiu d'aquest missatge és informar-nos de la posició del *marker*, el seu context, la incertesa del punt i l'id del *marker*. Amb això en ment, vaig crear el missatge *BetterPoint.msg*, el qual està format pels següents atributs:

```
#Id del marker vist
int32 id

#Posició x, y i z del marker
geometry_msgs/PointStamped position

#Desviació de x, y i z
geometry_msgs/PointStamped deviation
```

Amb aquest missatge ja podem tenir tot allò necessari per enviar les dades correctes al EKF-SLAM. No obstant, ens falta un petit detall, ja que els *markers* ens entren en forma de *array*, i, per tant, el nostre output hauria de ser també en forma de *array*. Per tant, és necessari crear un missatge més.

8.4.3 PointArray

Molt més simple que el missatge anterior, ja que només és necessari implementar una *array* d'aquest últim. Així doncs, vaig crear el missatge *PointArray.msg*, el qual està format pels següents atributs:

```
#Array de markers vistos
BetterPoints[] points
```

8.5 EKF-SLAM

Per finalitzar, haurem de dissenyar un node que, utilitzant l'odometria del robot i les dades publicades pel paquet ar_marker_corrector, ens publiqui la posició del robot filtrada utilitzant SLAM-EKM.

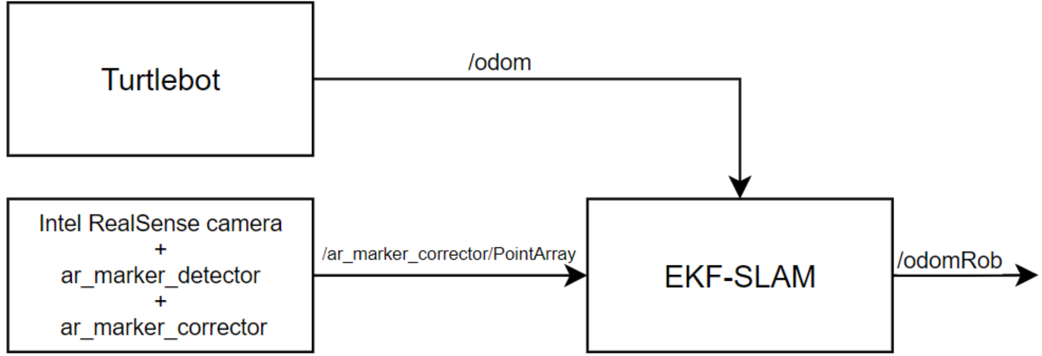


Figura 8.7: Diagrama de nodes ROS EKF-SLAM

En quant al patró utilitzat, es va decidir que la manera òptima era utilitzar una classe genèrica de EFK i després implementar classes que heretin d'ella.

8.5.1 ExtendedKalmanFilter

Primer de tot, abans de començar amb les herències, és necessari preparar les següents funcions:

8.5.1.1 prediction

$$\hat{x} = f(\hat{x}, u) \qquad P = APA^T + WQW^T$$

8.5.1.2 update

$$\hat{z} = y - h(\hat{x}) \qquad Z = HPH^T + R \qquad K = PH^T Z^{-1}$$

Un cop calculada la Z i la $\hat{z}$ hem de cridar compatibilityTest per saber si hem de realitzar les següents equacions.

$$x = \hat{x} + K\hat{z} \qquad P = P - KZK^T$$

Ara, ja podem seguir el diagrama de classes que ve a continuació i definir les classes que heretaran de la principal per dur a terme el EKF.

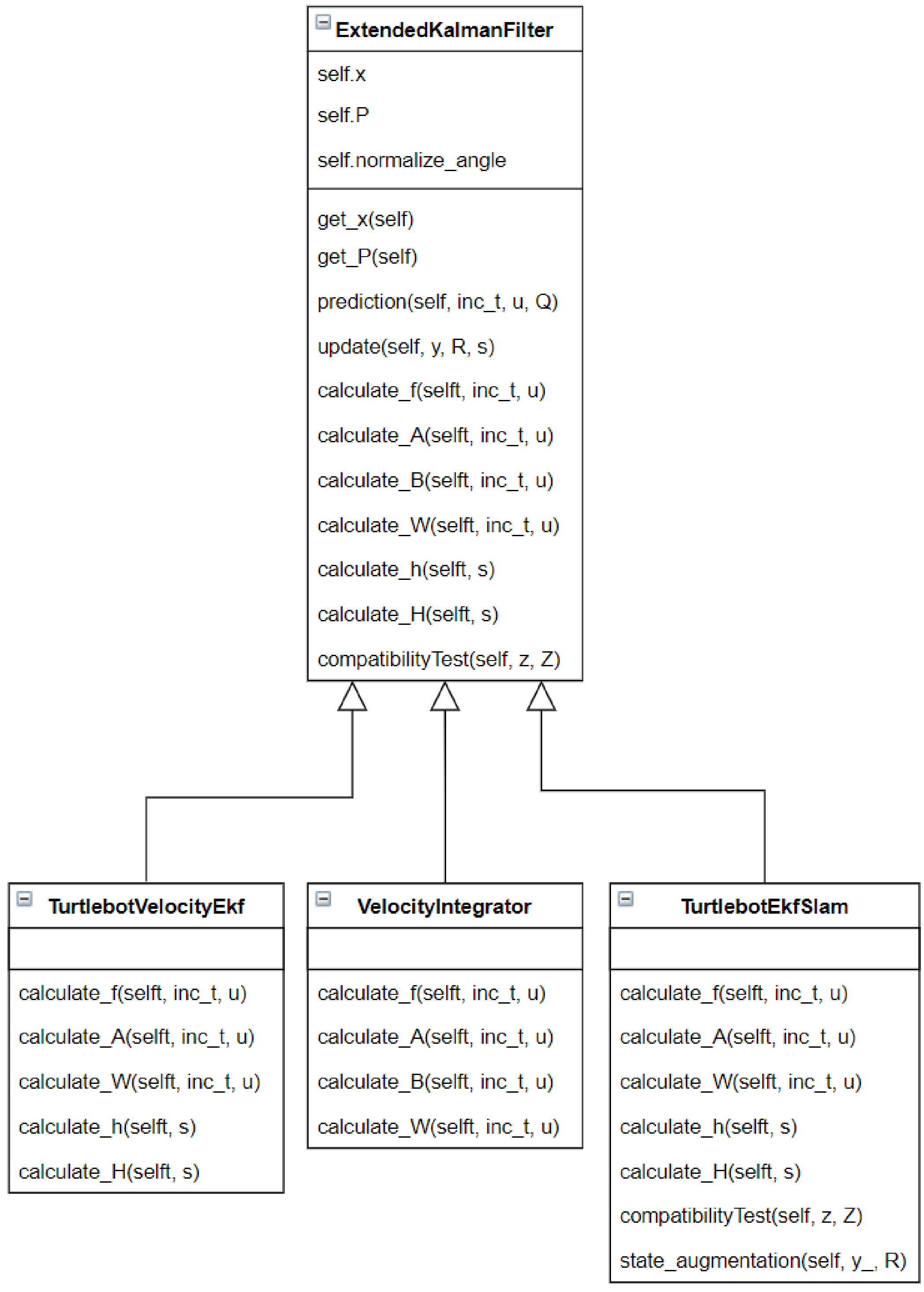


Figura 8.8: Diagrama de classes EKF-SLAM

Per cada una de les classes, s'hauran d'implementar diferents funcions, d'acord a les equacions del *paper*: SLAM with EKF d'en Joan Solà [Solà 2014].

8.5.2 TurtlebotVelocityEkf

Utilitzarem aquesta classe per estimar la velocitat del robot i la seva incertesa.

L'estat a estimar és el següent:

$$\hat{x} = [v_x \ v_y \ w]$$

On v_x és la velocitat en x, v_y és la velocitat en y, i w és la orientació.

Utilitzarem el model de moviment següent:

$$f(\hat{x}, u, n) = \begin{cases} v_x = v_x + n_{vx} t \\ v_y = v_y + n_{vy} t \\ w = w + n_w t \end{cases}$$

On $\hat{x}$ és l'estat, u és l'entrada del model, n és el soroll, i t eés l'increment de temps. Per simplificar el filtre, utilitzarem u com a buit.

Ara sí, ja podem preparar les equacions necessàries per a les següents funcions.

8.5.2.1 calculate_f

$f(\hat{x}, u) = A\hat{x} + Bu$

8.5.2.2 calculate_A

$A = \frac{\partial f(\hat{x},u,n)}{\partial \hat{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

8.5.2.3 calculate_W

$W = \frac{\partial f(\hat{x},u,n)}{\partial n} = \begin{bmatrix} t & 0 & 0 \\ 0 & t & 0 \\ 0 & 0 & t \end{bmatrix}$

8.5.2.4 calculate_h

$h(\hat{x}) = H\hat{x}$

8.5.2.5 calculate_H

$$H = \frac{\partial h(\hat{x})}{\partial \hat{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8.5.3 VelocityIntegrator

Un cop calculada la velocitat del robot i la seva incertesa per a la classe TurtlebotVelocityEkf, la utilitzarem, juntament amb aquesta classe i la predicció de la classe pare per a integrar la nostra posició.

L'estat a estimar és el següent:

$$\hat{x} = [\Delta_x \ \Delta_y \ \Delta_\psi]$$

On Δ_x és l'increment en x, Δ_y és l'increment en y, i Δ_ψ és l'increment en ψ.

Utilitzarem el model de moviment següent:

$$f(\hat{x},u,n) = \begin{cases} \Delta_x = \Delta_x + \cos(\Delta_\psi)(tv_x + \frac{t^2}{2}n_{vx}) - \sin(\Delta_\psi)(tv_y + \frac{t^2}{2}n_{vy}) \\ \Delta_y = \Delta_y + \sin(\Delta_\psi)(tv_x + \frac{t^2}{2}n_{vx}) + \sin(\Delta_\psi)(tv_y + \frac{t^2}{2}n_{v_y}) \\ \Delta_\psi = \Delta_\psi + tw + \frac{t^2}{2}n_w \end{cases}$$

On $\hat{x}$ és l'estat, u és l'entrada del model que ve donada per la classe TurtlebotVelocityEkf, n és el soroll, i t és l'increment de temps. Per simplificar el filtre, utilitzarem u com a buit.

Ara sí, ja podem preparar les equacions necessàries per a les següents funcions.

8.5.3.1 calculate_f

$f(\hat{x},u) = A\hat{x} + Bu$

8.5.3.2 calculate_A

$$A = \frac{\partial f(\hat{x},u,n)}{\partial \hat{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8.5.3.3 calculate_B

$$B = \frac{\partial f(\hat{x},u,n)}{\partial u} = \begin{bmatrix} \cos(\Delta_\psi)t & -\sin(\Delta_\psi)t & 0 \\ \sin(\Delta_\psi)t & \cos(\Delta_\psi)t & 0 \\ 0 & 0 & t \end{bmatrix}$$

8.5.3.4 calculate_W

$$W = \frac{\partial f(\hat{x},u,n)}{\partial n} = \begin{bmatrix} \cos(\Delta_\psi)\frac{t^2}{2} & -\sin(\Delta_\psi)\frac{t^2}{2} & 0 \\ \sin(\Delta_\psi)\frac{t^2}{2} & \cos(\Delta_\psi)\frac{t^2}{2} & 0 \\ 0 & 0 & \frac{t^2}{2} \end{bmatrix}$$

8.5.4 TurtlebotEkfSlam

Un cop calculat l'increment en la posició per a la classe VelocityIntegrator, l'utilitzarem, juntament amb aquesta classe i l'update de la classe pare per estimar la posició del robot i la dels *landmarks*.

L'estat a estimar és el següent:

$$\hat{x} = [x\ y\ \psi\ l_{xi}\ l_{yi}\ \dots\ l_{xn}\ l_{yn}]$$

On $[x\ y\ \psi]$ són la posició i orientació del robot i $[l_{xi}\ l_{yi}\ \dots\ l_{xn}\ l_{yn}]$ és la posició dels l'*landmarks*.

Utilitzarem el model de moviment següent:

$$f(\hat{x},u,n) = \begin{cases} x = x + \cos(\psi)(\Delta_x + n_x) - \sin(\psi)(\Delta_y + n_y) \\ y = y + \sin(\psi)(\Delta_x + n_x) + \cos(\psi)(\Delta_y + n_y) \\ \psi = \psi + (\Delta_\psi + n_\psi) \\ \vdots \\ l_{xn} = l_{xn} + n_{lxn} \\ l_{yn} = l_{yn} + n_{lyn} \end{cases}$$

On $\hat{x}$ és l'estat, u és l'entrada del model que ve donada per la classe VelocityIntegrator, n és el soroll, i t és l'increment de temps. Per simplificar el filtre, utilitzarem u com a buit.

Ara sí, ja podem preparar les equacions necessàries per a les següents funcions.

8.5.4.1 compatibilityTest

$M^2 > z^T Z^{-1} z$

8.5.4.2 calculate_f

$$
\begin{aligned}
aux_x &= x + \cos(\psi)\Delta_x - \sin(\psi)\Delta_y \\
aux_y &= y + \sin(\psi)\Delta_x + \cos(\psi)\Delta_y \\
aux_\psi &= \psi + \Delta_\psi
\end{aligned}
$$

$$
f(\hat{x}, u) = [aux_x \;\; aux_y \;\; aux_\psi \;\; l_{xi} \;\; l_{yi} \;\; \dots \;\; l_{xn} \;\; l_{yn}]
$$

8.5.4.3 calculate_A

$$
\mathbf{A} = \frac{\partial f(\mathbf{x},\mathbf{u})}{\partial \mathbf{x}} = \begin{bmatrix}
1 & 0 & -\sin(\psi)\Delta_x - \cos(\psi)\Delta_y & \dots & 0 & 0 \\
0 & 1 & \cos(\psi)\Delta_x - \sin(\psi)\Delta_y & \dots & 0 & 0 \\
0 & 0 & 1 & \dots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \dots & 1 & 0 \\
0 & 0 & 0 & \dots & 1 & 1
\end{bmatrix}
$$

8.5.4.4 calculate_W

$$
\mathbf{W} = \frac{\partial f(\mathbf{x},\mathbf{u},\mathbf{n})}{\partial \mathbf{n}} = \begin{bmatrix}
\cos(\psi) & -\sin(\psi) & 0 & \dots & 0 & 0 \\
\sin(\psi) & \cos(\psi) & 0 & \dots & 0 & 0 \\
0 & 0 & 1 & \dots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \dots & 1 & 0 \\
0 & 0 & 0 & \dots & 0 & 1
\end{bmatrix}
$$

8.5.4.5 calculate_h

$$
h(\mathbf{x}) = \begin{cases}
{}^{v}l_{ix} = -\cos(\psi)x - \sin(\psi)y + \cos(\psi)l_{ix} + \sin(\psi)l_{iy} \\
{}^{v}l_{iy} = \sin(\psi)x - \cos(\psi)y - \sin(\psi)l_{ix} + \cos(\psi)l_{iy}
\end{cases}
$$

8.5.4.6 calculate_H

$$
\mathbf{H} = \frac{\partial h(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix}
-\cos(\psi) & -\sin(\psi) & \sin(\psi)x - \cos(\psi)y - \sin(\psi)l_i x + \cos(\psi)l_{iy} & 0 \dots & \cos(\psi) & \sin(\psi) & \dots 0 \\
\sin(\psi) & -\cos(\psi) & \cos(\psi)x + \sin(\psi)y - \cos(\psi)l_i x - \sin(\psi)l_i y & 0 \dots & -\sin(\psi) & \cos(\psi) & \dots 0
\end{bmatrix}
$$

8.5.4.7 state_augmentation

$$G_R = \frac{\partial g(\hat{x},y)}{\partial \hat{x}[0:3]} = \begin{bmatrix} 1 & 0 & -\sin(\psi)^v l_{ix} - \cos(\psi)^v l_{iy} \\ 0 & 1 & \cos(\psi)^v l_{ix} - \sin(\psi)^v l_{iy} \end{bmatrix}$$

$$G_y = \frac{\partial g(\hat{x},y)}{\partial y} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix}$$

$$P_{LL} = G_R P[0:3, 0:3] {G_R}^T + G_y R {G_y}^T$$

$$P_{LR} = G_R P[0:3, :]$$

$$P^+ = \begin{bmatrix} P & {P_{LR}}^T \\ P_{LR} & P_{LL} \end{bmatrix}$$

$$g(\hat{x}, y) = \begin{cases} l_{ix} = x + \cos(\psi)^v l_{ix} - \sin(\psi)^v l_{iy} \\ l_{iy} = y + \sin(\psi)^v l_{ix} + \cos(\psi)^v l_{iy} \end{cases}$$

$$\hat{x}^+ = \begin{bmatrix} \hat{x} \\ g(\hat{x}, y) \end{bmatrix}$$

Capítol 9

Implementació i proves

9.1 Integració de la càmera Intel RealSense D435i

En aquest projecte volem substituir la Kinect que porta el Turtlebot 2 per a la càmera Intel RealSense. Per aconseguir-ho, és necessari seguir els següents passos:

9.1.1 Instal·lació en Ubuntu 20.04

```
# Registrem la clau pública del servidor
sudo apt-key adv --keyserver keys.gnupg.net --recv-
    key F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE ||
    sudo apt-key adv --keyserver hkp://keyserver.
    ubuntu.com:80 --recv-key
    F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE

# Afegim el servidor a la llista de repositoris
sudo add-apt-repository
"deb␣https://librealsense.intel.com/Debian/apt-repo␣
    focal␣main" -u

# Instal·lem les llibreries
sudo apt-get install librealsense2-dkms
sudo apt-get install librealsense2-utils
sudo apt-get install librealsense2-dev
sudo apt-get install librealsense2-dbg
```

Ara, verificarem la instal·lació connectant la càmera i executant la següent comanda:

```
realsense-viewer
```

Per a més informació sobre la instal·lació en Ubuntu feu clic aquí.

9.1.2 Calibratge

Per calibrar la càmera emprarem una utilitat proporcionada per Intel. [Int 021a]
Per instal·lar, hem de seguir els següents passos:

9.1.2.1 Instal·lació de les dependències de 3rs

```
# Ens assegurem que apt-get està actualitzat
sudo apt-get update

# Instal·lem les dependències
sudo apt-get install libusb-dev libusb-1.0-0-dev
sudo apt-get install libglfw3 libglfw3-dev
sudo apt-get install freeglut3 freeglut3-dev
```

9.1.2.2 Intal·lació del paquet mitjançant Amazon AWS

```
# Afegim el servidor a la llista de repositoris
echo 'deb␣http://realsense-hw-public.s3.amazonaws.com
   /Debian/apt-repo␣bionic␣main' | sudo tee/etc/apt/
   sources.list.d/realsense-public.list

# Registrem la clau pública del servidor
sudo apt-key adv --keyserver keys.gnupg.net --recv-
   key 6F3EFCDE

# Refresquem la llista de repositoris
# i paquets disponibles
sudo apt-get update

# Ja podem instal·lar l'eina de calibratge
sudo apt-get install librscalibrationtool
```

9.1.2.3 Execució de l'eina de calibratge

En primer lloc, abans d'iniciar el programa, necessitarem imprimir la imatge de referència pel calibratge, respectant les mides que s'indiquen. [Int 021a] Una vegada imprès correctament i col·locat sobre una superfície plana, ja podem iniciar el programa i seguir les instruccions que ens proporciona la GUI.

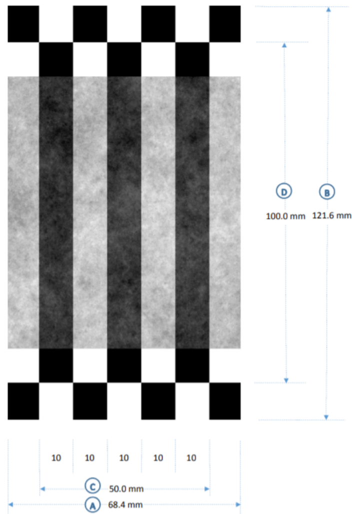


Figura 9.1: Objectiu de calibratge proporcionat per Intel

```
Intel.Realsense.DynamicCalibrator
```

9.1.3 Instal·lació en ROS Noetic

Una vegada tinguem els paquets del SDK2.0, és molt senzill instal·lar els paquets necessaris de ROS. [Int 021c].

```
#Creem una variable amb el nom de la distribució
export ROS_VER=noetic

# Instal·lem els paquets de ROS
sudo apt-get install ros-$ROS_VER-realsense2-camera

# Aquest paquet és opcional
sudo apt-get install ros-$ROS_VER-realsense2-
    description
```

9.2 Integrar un nou punt d’accés als robots Turtlebot 2

9.2.1 Intal·lació del router

La instal·lació del router no representa una gran dificultat. La gran majoria tenen una interfície gràfica senzilla i entenedora.
Per començar, en connectarem via ethernet i escriurem al navegador 192.168.31.1. Seguidament, s’ens obrirà la pagina inicial de configuració. Un cop acceptats els termes i condicions, escollirem el nom i la contrasenya del Wi-Fi i ja tindrem el router funcional. Arribats fins aquí, només quedarà un petit detall per a facilitar la connexió als alumnes, i és que haurem d’entrar a la configuració del router i la configurarem amb una IP fixe al portàtil del Turtlebot 2 a la 192.168.31.2.

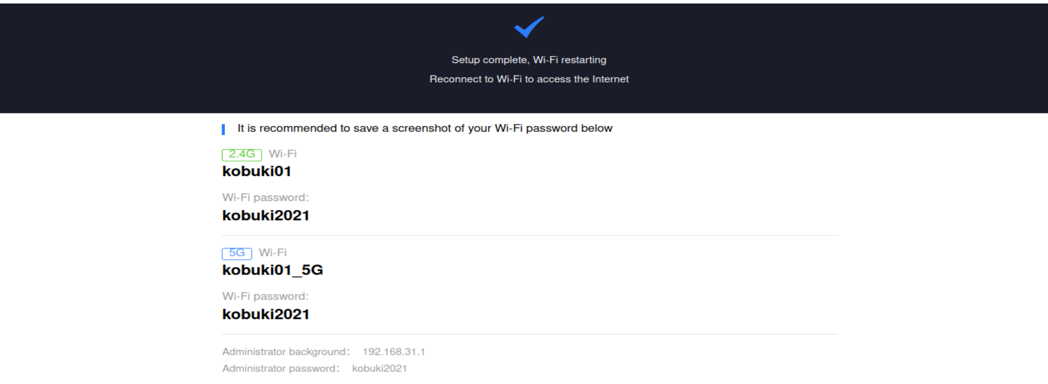


Figura 9.2: Configuració del Wi-Fi

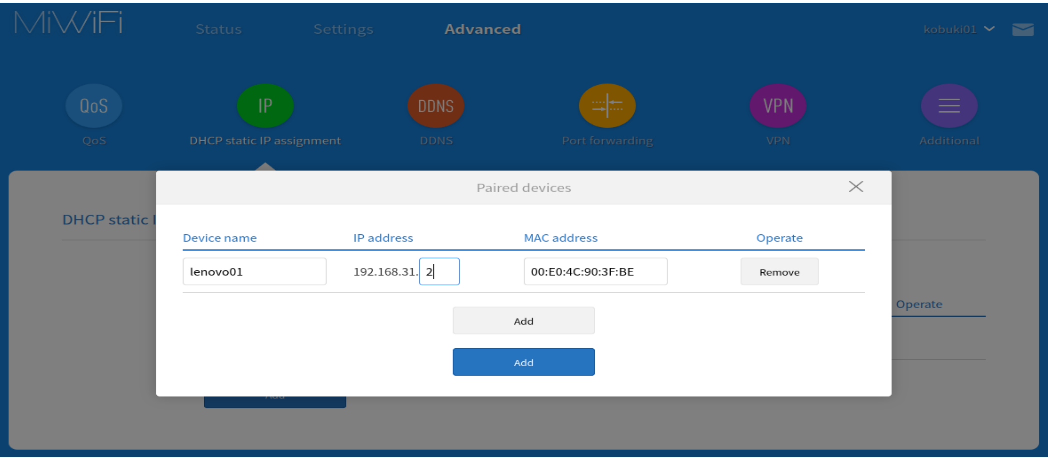


Figura 9.3: Configuració de l’IP fixe

9.2.2 Manual de connexió

Per poder connectar-nos des de qualsevol PC al Turtlebot 2 haurem de fer el següent: Primer, ens connectem al Wi-Fi del robot que volem controlar (ex: Kobuki01). Un cop connectats, hem de fer SSH al portàtil del Turtlebot 2 i després hem d'anar al nostre fitxer de hosts (ex: sudo nano /etc/hosts) i afegir la següent línia: 192.168.31.2 lenovo0X on X és el número de Turtlebot 2 on ens connectem.

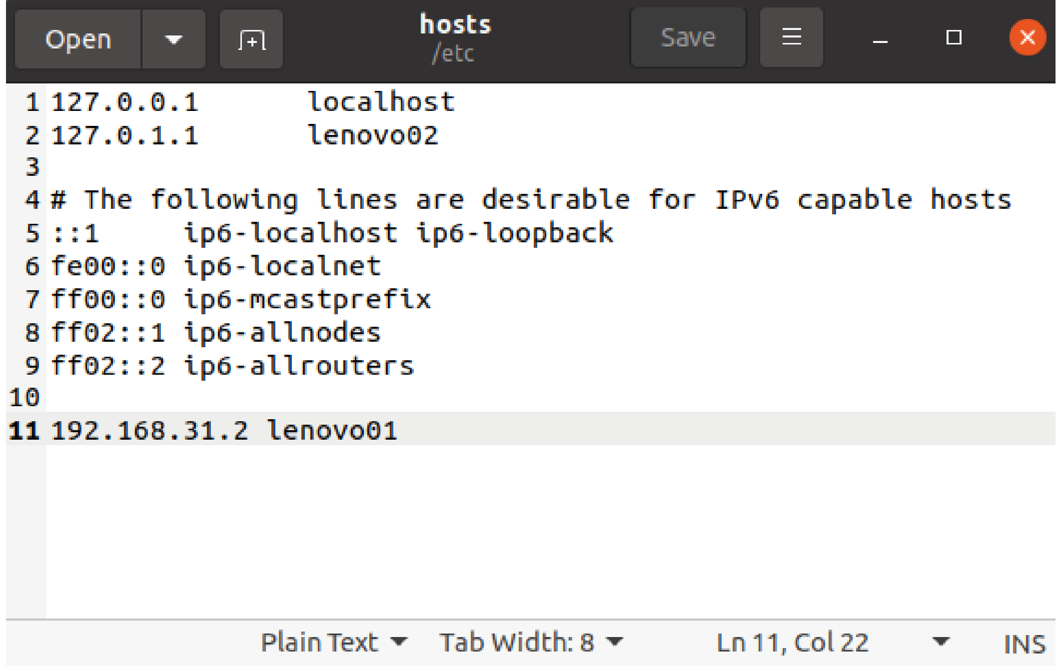


Figura 9.4: Mostra de configuració del fitxer hosts

En aquest exemple, es pot veure com des del nostre portàtil —en aquest cas, el lenovo02— ens connectem al lenovo01 per tal de poder controlar-lo de manera remota.

9.3 Integració del paquet aruco_ros

Primer de tot, hem de descarregar la llibreria ArUco, que podem obtenir de la pàgina web oficial aquí. Un cop descarregada, descomprimim el ZIP, obrim un terminal dins del directori i executem la comanda *make*. [ArU 021]

Un cop instal·lat, vam observar que no podíem executar correctament les utilitats d'ArUco. Vam estar investigant fins que vam trobar que la llibreria ArUco treballa amb una versió de OpenCV que no és compatible amb Python 3. Per solucionar el problema, hem utilitzat un paquet de ROS addicional. Aquest paquet el descarregarem del BitBucket i el copiarem al nostre *workspace* i després el compilarem.

Ja podem utilitzar la llibreria ArUco i detectar els *markers* del nostre entorn.

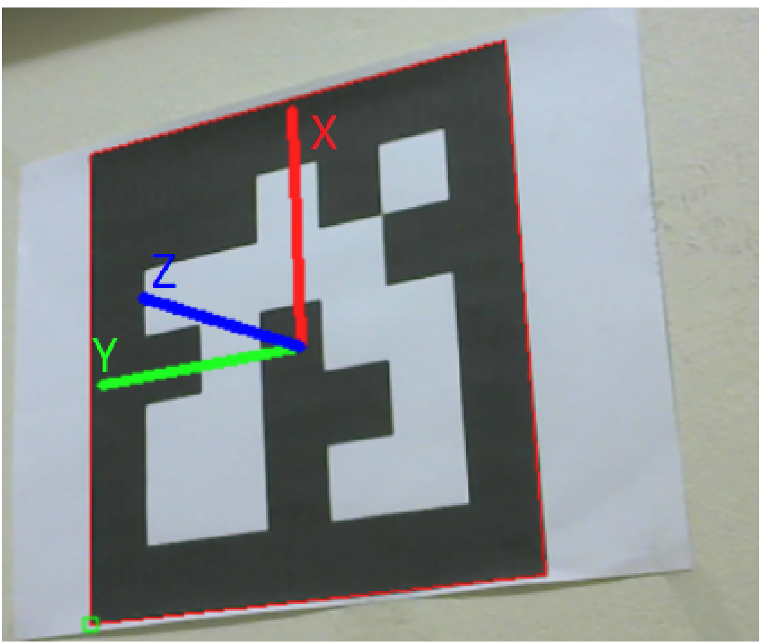
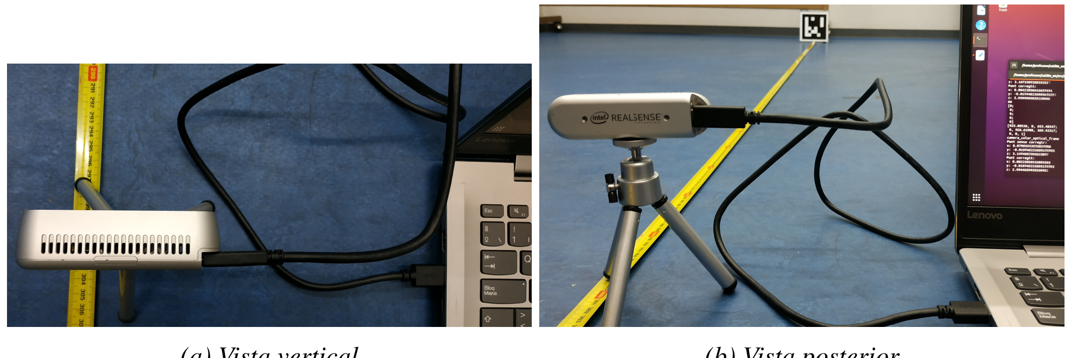


Figura 9.5: Detecció de marker amb ArUco

9.4 Error amb el càlcul de distàncies

Amb tots els paquets i *drivers* ja instal·lats vam començar a fer les primeres proves agafant mesures. Durant aquest procés, ens van adonar que hi havia un error en la mesura de les distàncies. Aquest error és feia molt obvi en les grans distàncies. Aproximadament, es tractava d'un error del 5% en l'eix Z. Primer, ho vam atribuir a un error de calibratge, però deprès de diversos intents de calibratge utilitzant diferents mètodes, vam decidir seguir avançant amb el projecte i corregir l'error amb una nova capa de calibratge.

Com a demostració de l'error i amb l'objectiu de preparar un script que el corregeixi, hem construït un entorn on coneixem en tot moment la distància del *marker* respecte a l'objectiu de la càmera. Així doncs, vam muntar un *marker* fixe i amb una cinta mètrica vam col·locar la càmera a 3 m exactes.



(a) Vista vertical *(b) Vista posterior*

Figura 9.6: Col·locació de la càmera

Podem apreciar que l'objectiu de la càmera es troba a uns 3 m. Ignorem l'error que pugui tenir la cinta mètrica així com la pròpia col·locació de la càmera, ja que es tractaria de mil·límetres.

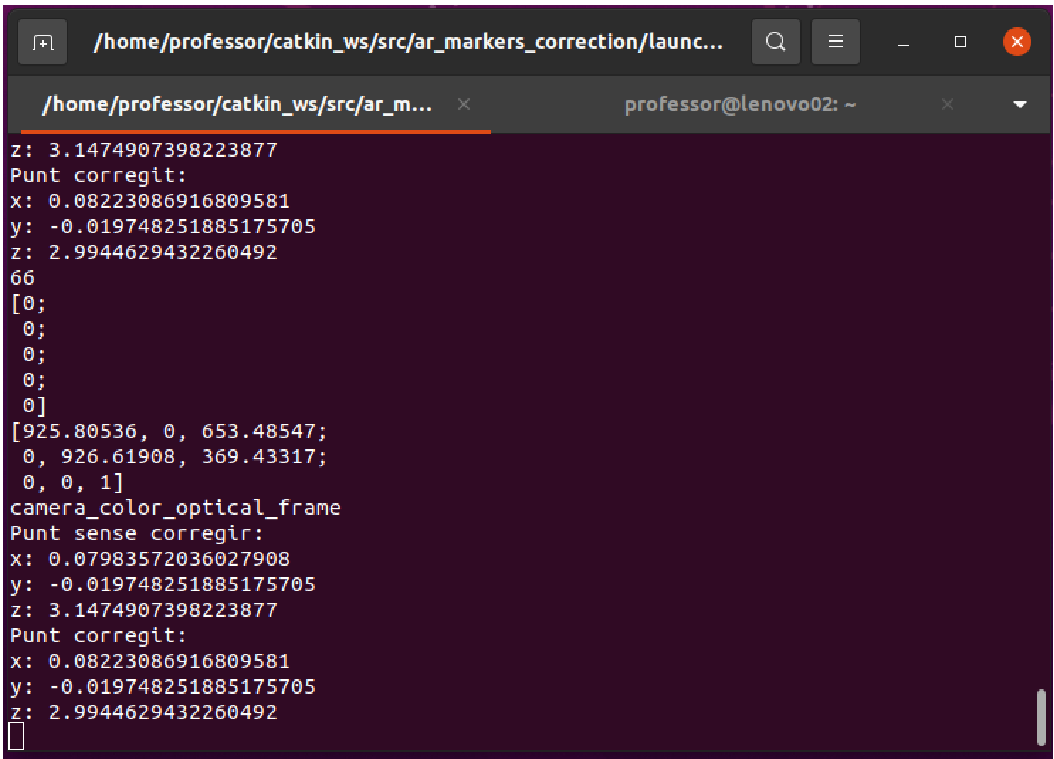


Figura 9.7: Captura de pantalla del portàtil amb les lectures de la càmera (terminal)

Fent referència a la figura 9.6, ampliem la imatge per poder llegir els valors. Clarament tenim una gran desviació, ja que ens marca 3,14 m però segons la cinta mètrica estem a uns 3 m. Tant el valor de la Z com el de la X necessiten ser corregits pel correcte funcionament del EKF-SLAM.

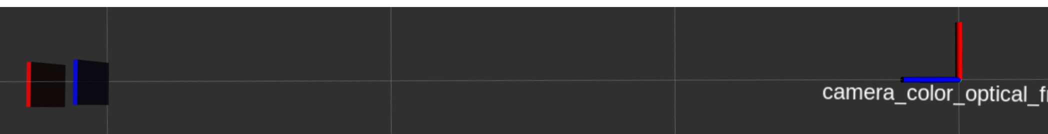


Figura 9.8: Captura de pantalla del portàtil amb les lectures de la càmera (rviz)

Observant l'rviz som capaços de veure que clarament les diferències entre la mesura de la càmera (vermell) i la lectura "real"(blau) són molt diferents.

9.5 Obtenció d'un dataset

Per tal de poder treballar amb l'algoritme sense dependre del món real, ROS ens permet guardar en un fitxer amb totes les dades suficients per després poder simular aquest mateix.

9.5.1 Construcció d'un entorn real

Primer, començarem per construir el nostre entorn. De manera una mica rudimentària, hem utilitzat taules i caixes del CIRS per construir-lo. La intenció és crear un entorn tancat on el robot es pugui moure sense problema, però que també disposi d'obstacles a esquivar i de racons amb poca visibilitat.

Figura 9.9: Entorn construït

9.5.1.1 Col·locació de les ArUco

Un cop construït, s’han de col·locar els diferents *markers* ArUco. La idea és que el robot no vegi *markers* constantment, sinó que se’ls trobi pel camí i després els perdi de vista.

(a) Marker 20 a la posició (13,0)

(b) Marker 17 a la posició (123,60)

(c) Marker 19 a la posició (276,76)

(d) Marker 18 a la posició (104,187)

(e) Marker 21 a la posició (0,268)

(f) Marker 22 a la posició (245,273)

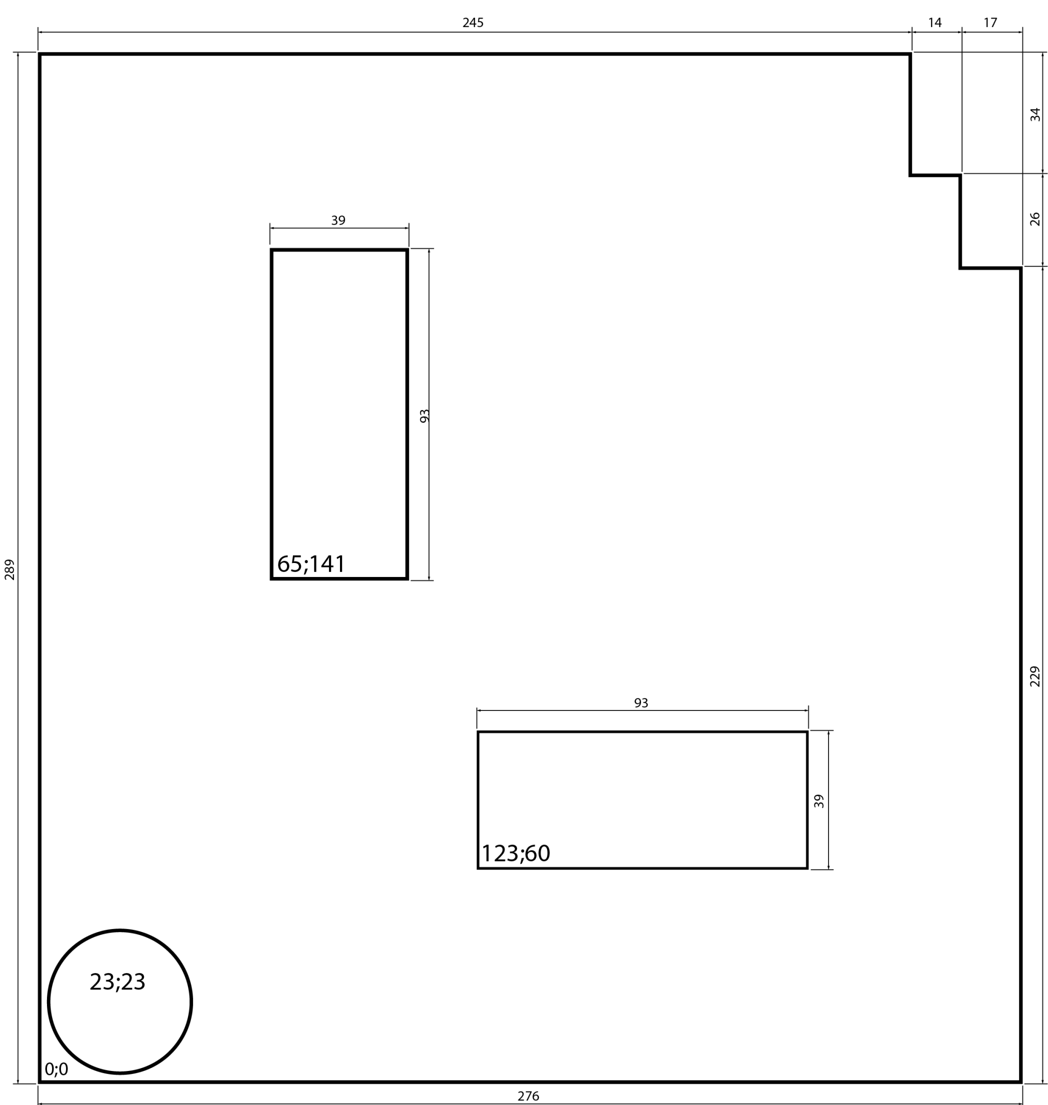


Figura 9.11: Plànol de l'entorn construït

Aquest és el plànol resultant de la construcció de l'entorn, però vam decidir crear un altre mapa, aquesta vegada al laboratori de robòtica.

Per problemes amb la detecció de ArUco en aquell mapa, al final vaig decidir descartar-ne l'ús i només utilitzar aquest.

9.5.2 Construcció d’un entorn simulat

Utilitzant el gazebo he modelat un entorn simulat molt semblant a l’entorn real del CIRS, aquest entorn busca crear un recorregut on no rebem cap tipus de soroll extern al robot. Així, es pot observar de manera més clara el funcionament de l’algoritme de localització. Per crear aquest entorn ha estat necessari l’ús de multituds de fonts, tant per crear el món, com cadascun dels models que hi ha dins. [Gaz 2021] [Construct 2021] [URD 2020] [Nievas 021] [spa 2018]

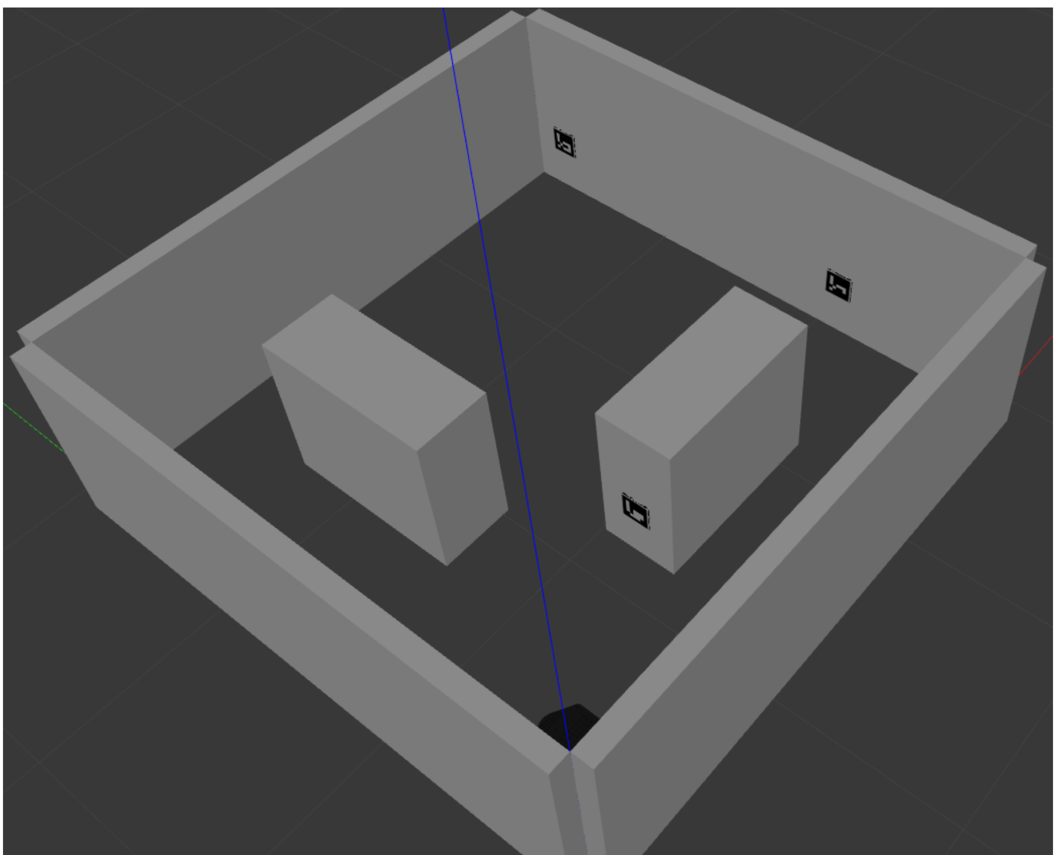
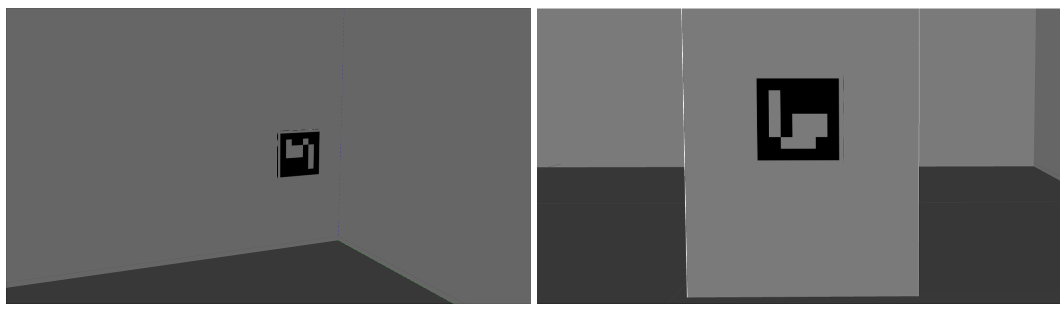


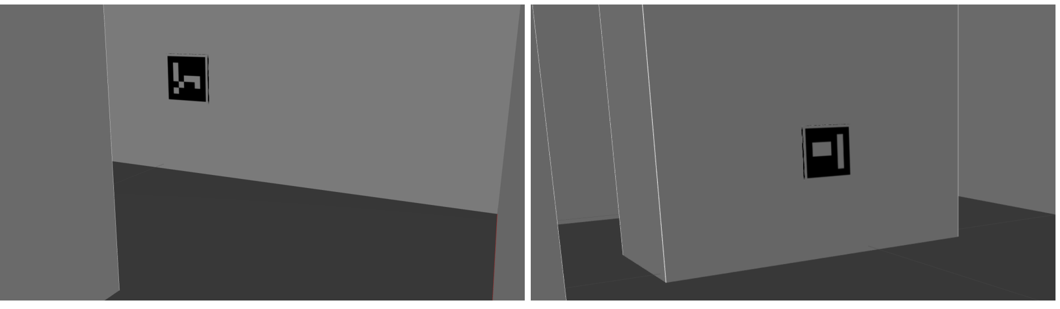
Figura 9.12: Entorn modelat amb gazebo

9.5.2.1 Col·locació de les ArUco

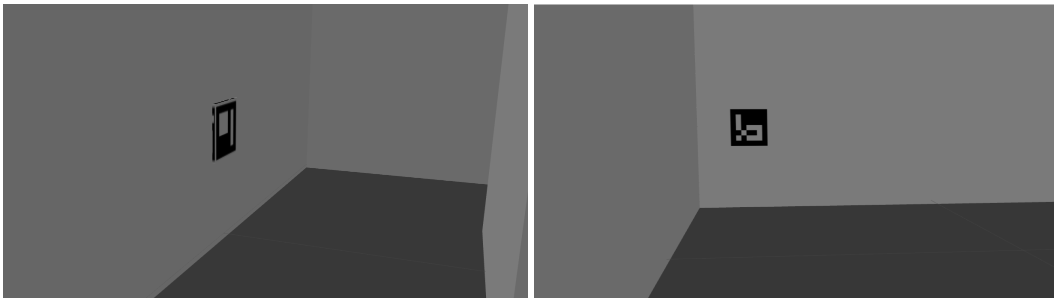
En aquest cas, col·locar les ArUco no és tan senzill com imprimir un paper i enganxar-lo, sinó és necessari crear un model 3D amb blender i col·locar-li la imatge del *marker* com a textura per després afegir-lo a l'entorn. Afortunadament, vaig trobar un projecte de GitHub [Nievas 021] que et generava els models que volguessis i que es pot trobar a l'apartat 13.4.



(a) Marker 22 a la posició (15,0) *(b) Marker 23 a la posició (123,80)*



(c) Marker 24 a la posició (289,90) *(d) Marker 20 a la posició (104,187)*



(e) Marker 21 a la posició (0,200) *(f) Marker 25 a la posició (289,270)*

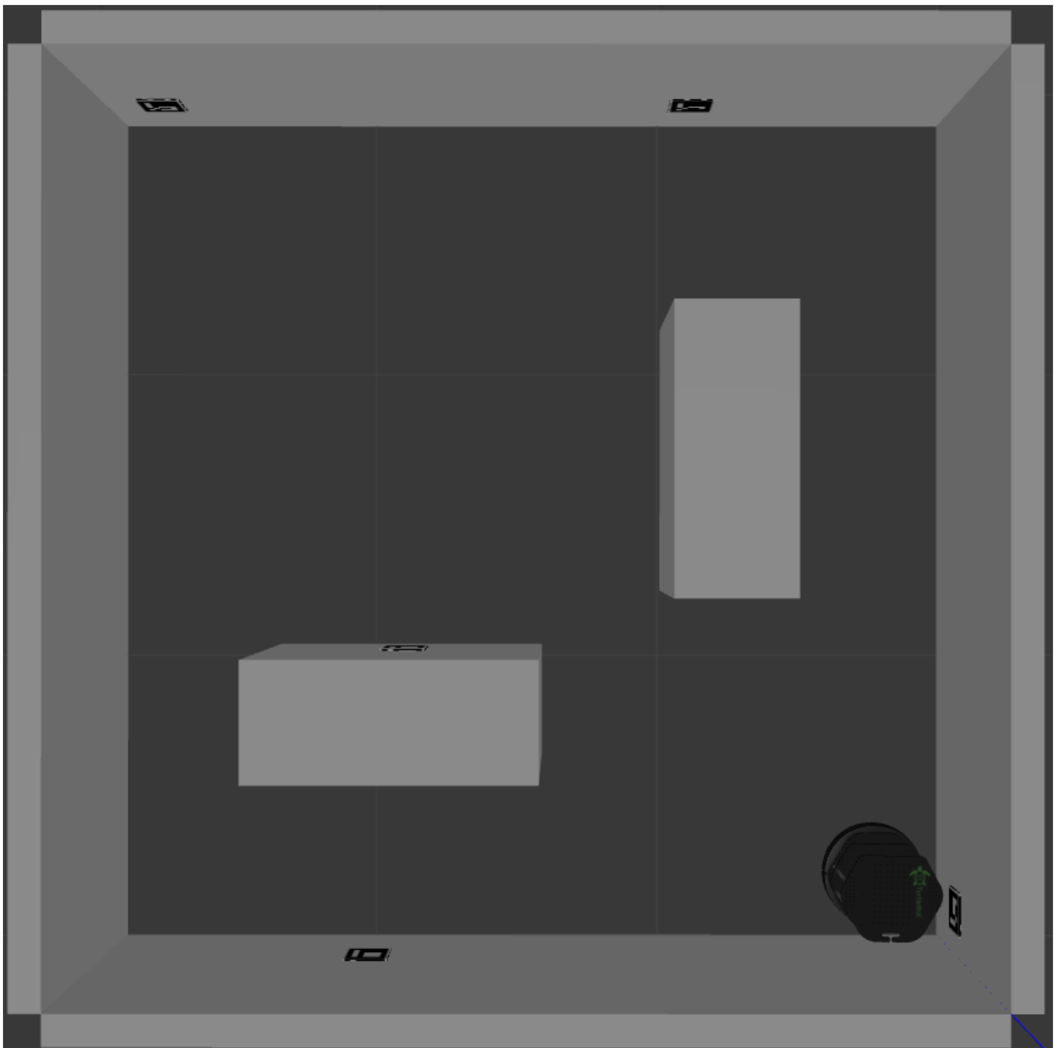


Figura 9.14: Vista aèria de l'entorn simulat

Tot i que vaig modelar diversos entorns, finalment vam decidir utilitzar només aquest, per la similitud amb el real, i deixar els altres per a futurs alumnes que vulguin realitzar els seus recorreguts simulats sense necessitat de crear-los ells.

9.5.3 Enregistrar el recorregut per l'entorn

Per registrar totes les dades dels sensors durant el recorregut utilitzarem la comanda rosbag, concretament:

```
#Registrem tots els topics
rosbag record -a
```

De moment registrarem, tots els tòpics per estar segurs de no deixar-nos-en cap.

Un cop executada la comanda, podem començar a moure el robot per el mapa, mentre tant, el rosbag s'encarrega de guardar totes les dades publicades pel ROS. Finalitzat el recorregut, podem aturar amb ctrl+c el rosbag i ja tenim el nostre paquet, que podem reproduir en qualsevol moment.

Per reproduir el recorregut utilitzarem la següent comanda:

```
rosbag play <nom del fitxer>
```

9.6 Creació de ar_marker_corrector

Una vegada tenim tots els paquets integrats i el nou dispositiu instal·lat i calibrat, necessitem un node de ROS que llegeixi aquest sensor i ens proporcioni les dades que necessitem per tal de fer SLAM. [Construct 2021] [ROS 021]

9.6.1 Crear el paquet de ROS

Crear un paquet de ROS és molt senzill, només es necessita la següent comanda:

```
catkin_create_pkg <package_name> [depend1] [depend2]
    [depend3]

#En aquest cas, la nostra comanda quedaria així:
catkin_create_pkg ar_mar std_msgs rospy
```

Després d'executar aquestes comandes, haurem creat el paquet amb les dependències bàsiques. Ara toca definir el msg, src i launch del nostre paquet.

9.6.2 Definir els missatges a publicar (msg)

Abans de programar res, hem de decidir quins missatges publicarà el nostre paquet, per tant, crearem la carpeta msg i, dins, hi crearem un fitxer de tipus .msg. En el nostre cas són necessaris dos .msg. El primer, anomenat BetterPoints.msg, estarà format per 3 atributs:

```
#Id del marker vist
int32 id

#Posicio x, y i z del marker
geometry_msgs/PointStamped position

#Desviacio de x, y i z
geometry_msgs/PointStamped deviation
```

El segon, anomenat PointArray.msg, estarà format per 1 atribut:

```
#Array de punts vistos
BetterPoints[] points
```

Un cop creats el missatges, haurem de registrar-los, per això, anirem al document cmake.txt i buscarem l'apartat *add_message_files* i li afegirem els nostres missatges.

```
add_message_files(
 FILES
 BetterPoints.msg
 PointArray.msg
)
```

9.6.3 Crear el programa que corregeixi i publiqui les dades (src)

Primer, crearem la carpeta src on hi posarem tots els fitxers Python que necessitem. En el nostre cas, només en farem servir un: dataGetter.py.

En aquest fitxer necessitem posar tota la lògica necessària per subscriure'ns i publicar els tòpics pertinents.

Com es pot observar en el codi, ens subscrivim al tòpic "/ar_detector/ar_markers"on rebrem tota la informació sobre el *marker* que estiguem observant. Rebuda la informació, primer la filtrarem segons si acceptem les distàncies i l'Id del *marker*. Després, tractarem la posició per corregir-la amb el polinomi que hem trobat tal i com s'explicarà a continuació a la Subsubsecció 9.6.3.1 i la transformarem del TF de la càmera al TF del robot. Amb la posició corregida i la incertesa calculada, ja podem publicar el missatge que hem creat a la Subsecció 9.6.2.

9.6.3.1 Re-calibratge de la càmera

Per corregir l'error de la càmera comentat a la Secció 9.4 he creat aquest petit i simple script que es dedica a captar les dades de mesura i guardar-les en un fitxer csv amb el següent format:

xReal	zReal	x		z	
		mean	stdev	mean	stdev
1.2	2	1.259745	0.002552	2.133567	0.004443

Figura 9.15: Resultats de mesurar el marker des de X = 1.2 i Z = 2

9.6.3.1.1 Obtenció dels punts

Per obtenir els punts és necessari fer un petit estudi del camp de visió de la càmera. Vaig determinar que l'angle de visió de la càmera era d'uns 62º. Amb aquestes dades vaig crear la distribució (en cm) següent:

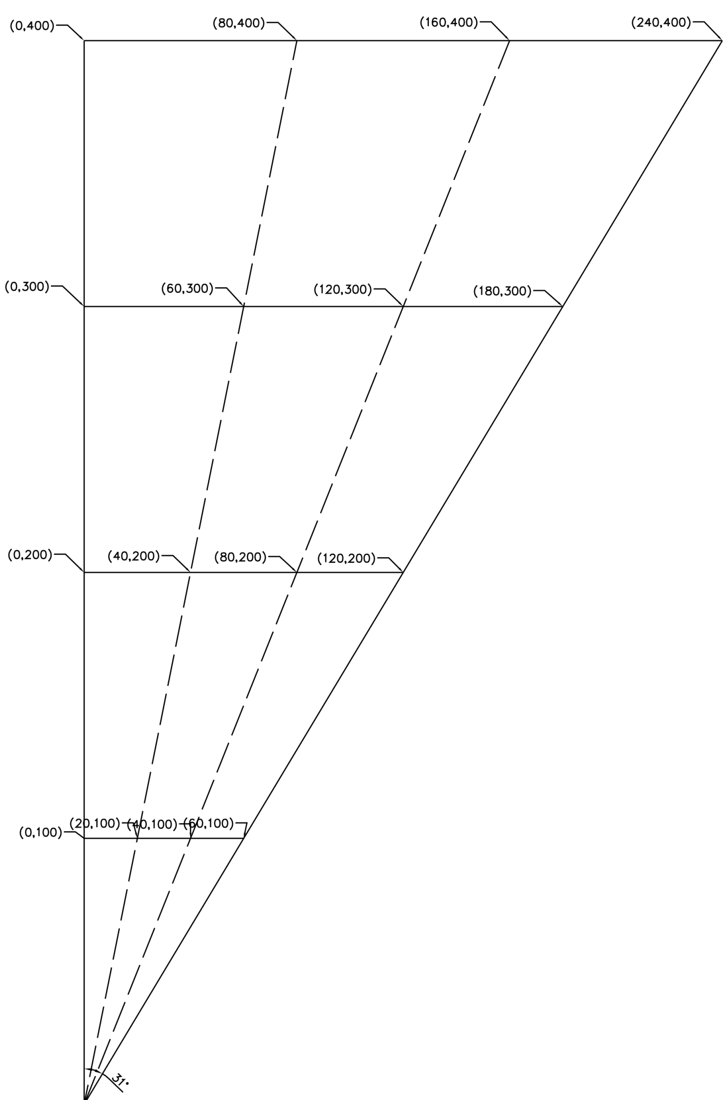


Figura 9.16: Mapa de punts

Com es pot observar, tenim 16 punts on col·locar el *marker* de forma precisa. Per manca d'espai i per reduir el nombre de mesures, es mesura el costat dret i s'extrapolen les dades a l'altre costat, suposant el cas ideal en el qual la càmera obté les dades per igual pels dos costats.

9.6.3.1.2 Obtenció de totes les dades

Una vegada sabem els punts a utilitzar, es tracta de moure, amb gran precisió, el nostre *marker* per tots els punts i utilitzar l'script de *Python* per obtenir les lectures corresponents.
Finalment, tindrem 16 fitxers que caldria agrupar en un de sol. A més, volem les dades equivalents pel costat esquerre. Per això, utilitzarem un altre script que recorre tots els fitxers, extreu les mesures de X i Z (real i mesurada), la duplica i nega la X (per tal de generar lectures del costat esquerre), i les fusiona en un sol fitxer de la següent forma:

Xreal	Zreal	x	z
-2.4	4	-2.607059	4.4303079
-1.8	3	-1.8894345	3.19928091
-1.6	4	-1.7180439	4.39648419
-1.2	2	-1.2597447	2.1335672
-1.2	3	-1.2333572	3.14793528
-0.8	2	-0.8180874	2.08423762
-0.8	4	-0.8157113	4.29808195
-0.6	3	-0.6004685	3.15350735
-0.6	1	-0.6093078	1.02673494
-0.4	2	-0.3987392	2.09114874
-0.4	1	-0.4023841	1.01943656
-0.2	1	-0.1981144	1.02330712
0	4	-0.0382809	4.16870022
0	2	-0.0170896	2.07834302
0	3	-0.0273725	3.18137107
0	1	-0.0065674	1.01817894
0.2	1	0.19811435	1.02330712
0.4	2	0.3987392	2.09114874
0.4	1	0.40238412	1.01943656
0.6	3	0.60046845	3.15350735
0.6	1	0.60930775	1.02673494
0.8	2	0.81808738	2.08423762
0.8	4	0.81571132	4.29808195
1.2	2	1.25974474	2.1335672
1.2	3	1.23335718	3.14793528
1.6	4	1.71804385	4.39648419
1.8	3	1.8894345	3.19928091
2.4	4	2.60705896	4.4303079

Figura 9.17: Dades finals de calibratge

9.6.3.1.3 Creació de un pla polinòmic

Amb les dades del darrer apartat, ja és possible crear un pla polinòmic que s'ajusti a les nostres mesures i, per tant, que corregeixin l'error de la càmera. Per fer això, utilitzarem la X i Z mesurades com a punts X i Y en un pla. després, utilitzarem el Matlab i la funció "fit" per determinar el pla que, donada un X i Z (amb error), ens retorni una X rectificada i un altre pla que ens rectifiqui la Z.

```
fx=fit(a,Xr,'poly23')
plot(fx,a,Xr)
figure
fz=fit(a,Zr,'poly23')
plot(fz,a,Zr)
```

Una vegada executat aquest petit script, obtenim les funcions que determinen els dos plans i que ens corregiran l'error de mesura.

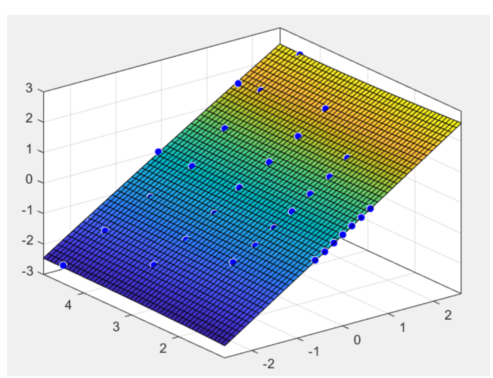


Figura 9.18: Pla corrector de X

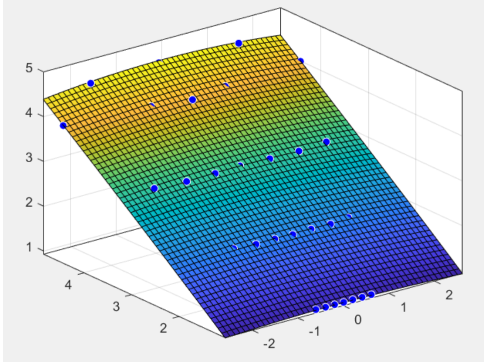


Figura 9.19: Pla corrector de Z

Aquests plans estan definits per les següents equacions:

```
Linear model Poly23:
fx(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y
        + p02*y^2 + p21*x^2*y + p12*x*y^2 + p03*y^3
Coefficients (with 95% confidence bounds):
  p00 =    -0.01141  (-0.1757, 0.1529)
  p10 =       1.036  (0.9161, 1.155)
  p01 =     0.01859  (-0.2086, 0.2457)
  p20 =    0.003187  (-0.05075, 0.05713)
  p11 =    -0.04202  (-0.1183, 0.03431)
  p02 =    -0.00806  (-0.09669, 0.08057)
  p21 =    -0.00182  (-0.01473, 0.01109)
  p12 =     0.00473  (-0.00672, 0.01618)
  p03 =    0.001254  (-0.009079, 0.01159)
```

Figura 9.20: Pla corrector de X

```
Linear model Poly23:
fz(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y
        + p02*y^2 + p21*x^2*y + p12*x*y^2 + p03*y^3
Coefficients (with 95% confidence bounds):
  p00 =     0.01062  (-0.1785, 0.1998)
  p10 =   0.0008652  (-0.1367, 0.1385)
  p01 =      0.9775  (0.716, 1.239)
  p20 =   -0.007451  (-0.06955, 0.05465)
  p11 =  -0.0006507  (-0.08852, 0.08722)
  p02 =    -0.01042  (-0.1125, 0.09161)
  p21 =   -0.003513  (-0.01838, 0.01135)
  p12 =   0.0001064  (-0.01308, 0.01329)
  p03 =   0.0003376  (-0.01156, 0.01223)
```

Figura 9.21: Pla corrector de Z

Aquests plans seran els encarregats de rebre com a *inputs* la X i la Z, i retornar la X i la Z corregides respectivament.

9.6.4 Crear el *laucher* del paquet (launch)

Primer, crearem la carpeta launch i, dins, hi crearem un fitxer .launch

```
<launch>
  <include file="$(find␣ar_markers_detector)/launch/
    detector.launch"/>
  <node name="ar_markers" pkg="ar_markers_correction"
    type="dataGetter.py" output="screen"></node>
</launch>
```

Com es pot observar en el fitxer de *launch* cridem un paquet extern, en aquest cas, el detector de *markers* ArUco i un fitxer de Python que hem creat.

Un cop finalitzat, és necessari moure'ns al *root* del *workspace* i compilar.

9.7 Estimació de la incertesa

Com és habitual, qualsevol mesura va lligada a una incertesa. Per això, utilitzant un mètode similar a la de la Subsubsecció 9.6.3.1, he calculat una corba polinòmica que ens determini la incertesa de cada punt.

9.7.1 Obtenció de totes les dades

Utilitzant punts aleatoris i el mateix script que a la Subsubsecció 9.6.3.1, farem que reculli 100 mesures per cada punt. Així doncs, per cada punt crearem una taula amb l'X o la Z i la desviació estàndard de les 100 mesures.

X	StDev
0	0
0.5	0.0037
1	0.0067
1.5	0.0088

Z	StDev
0	0
1	0.007
2	0.018
3	0.029
4	0.0424

Figura 9.22: Dades finals d'incertesa

9.7.2 Creació d'una corba polinòmica

Amb les dades obtingudes i utilitzant les mateixes comandes de l'apartat anterior, però amb una petita variació, podem calcular la corba que determini la nostra incertesa segons el punt on ens trobem.

```
valXx = [0;0.5;1;1.5];
valXy = [0;0.0037;0.0067;0.0088];

valZx = [0;1;2;3;4];
valZy = [0;0.007;0.018;0.028;0.0424];

fx=fit(valXx,valXy,'poly2')
plot(fx,valXx,valXy)
figure
fz=fit(valZx,valZy,'poly2')
plot(fz,valZx,valZy)
```

Una vegada executat aquest petit script, obtenim les funcions que determinen les dues corbes i que ens donen la incertesa segons el punt on ens trobem.

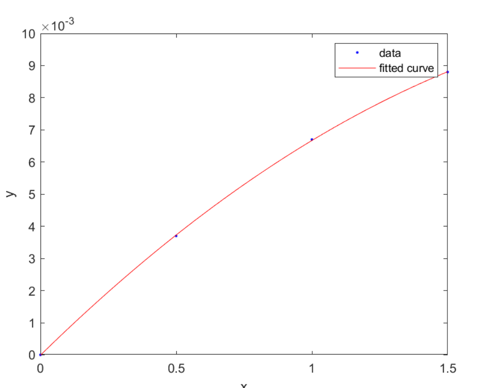


Figura 9.23: Funció d'incertesa en X

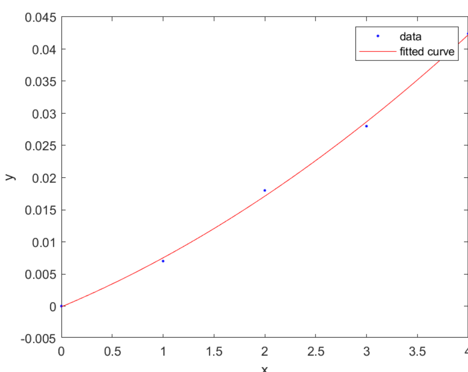


Figura 9.24: Funció d'incertesa en Z

```
Linear model Poly2:
  fx(x) = p1*x^2 + p2*x + p3
  Coefficients (with 95% confidence bounds):
   p1 = -0.0016 (-0.002736, -0.0004635)
   p2 = 0.00828 (0.006501, 0.01006)
   p3 = -1e-05 (-0.0005639, 0.0005439)
```

```
Linear model Poly2:
  fz(x) = p1*x^2 + p2*x + p3
  Coefficients (with 95% confidence bounds):
   p1 = 0.0009857 (-3.025e-05, 0.002002)
   p2 = 0.006637 (0.002399, 0.01088)
   p3 = -0.0001086 (-0.003686, 0.003469)
```

Com podem observar, les dues funcions poden donar valors negatius. Per tal d'anar sobre segur, tractarem els dos valors de p3 com a 0.

9.8 Implementació de les equacions EKF-SLAM

Amb tot els paquets ja implementats, ara toca implementar el EKF-SLAM. Per fer-ho utilitzarem com a referència les equacions del paper: SLAM with EKF d'en Joan Solà [Solà 2014] com s'ha explicat a l'apartat 10.2.2.

Degut a que la implementació en Python d'aquestes equacions pot ser utilitzada per resoldre la pràctica, que és fruit d'aquest treball, haure d'ometre el codi. Aixo si, podeu observar l'estructura sense les funcions implementades a la Secció 13.3

Capítol 10

Implantació i resultats

10.1 Implantació

Ja que el projecte es basa en ROS, la implantació és molt senzilla.

Primer, necessitem un workspace ROS. Assumint que tenim ROS instal·lat correctament, executem la següent comanda:

```
source /opt/ros/<rosVersion>/setup.bash
```

Després crearem el workspace

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

Amb el workspace creat, ja podem descarregar el paquet ar_markers_corrector i copiar-lo dins la carpeta catkin_ws/src. Un cop copiat, tornem a compilar el workspace

```
catkin_make
```

I, així doncs, ja podem utilitzar el paquet des de qualsevol terminal utilitzant la següent comanda:

```
roslaunch ar_markers_corrector corrector.launch
```

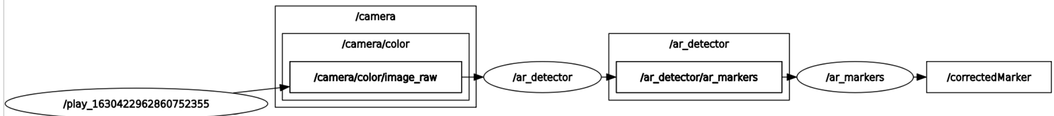


Figura 10.1: Topics actius al executar el launch

Al executar aquest paquet podem observar com s'inicia el /ar_detector, que es subscriu a la càmera per tal de publicar els /ar_markers, els quals són traduïts per aquest paquet i publicats en forma de /correctedMarker.

Pel que fa la EFK, només cal descarregar-lo i obrir un terminal dins el directori i executar:

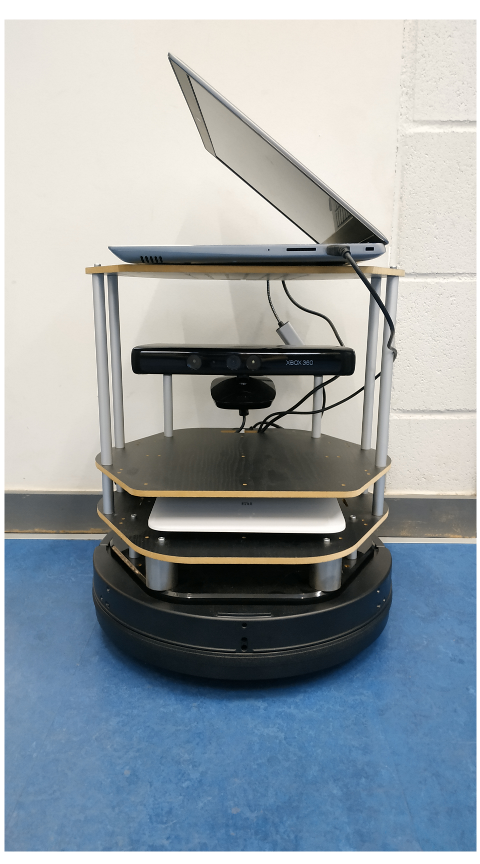
```
python turtlebotEKF.py
```

10.2 Resultats

10.2.1 Robot muntat i totalment equipat

Com a resultat d'aquest treball s'han muntat dos robots diferents.

Un d'ells, equipat amb el sensor antic de la Kinect i renovat amb un router que serà utilitzat en les pràctiques pels alumnes de la UdG.



(a) Vista anterior

(b) Vista posterior

Figura 10.2: Robot de pràctiques

L'altre, equipat amb el sensor nou Intel RealSense i també amb el router, s'ha utilitzat durant tota la realització d'aquest projecte i té el potencial de poder seguir usant-se també en pràctiques futures.

(a) Vista anterior *(b) Vista posterior*

Figura 10.3: Robot de TFG

10.2.2 EKF-SLAM

Amb el EKF-SLAM implementat amb Python, és hora de posar-lo a prova. Ens interessa que el filtre corregeixi correctament la posició del robot fent ús dels *landmarks*. Per això, utilitzant els dos entorns creats, explicats a l'apartat 9.7, executarem els rosbags registrats donant una volta pels entorns i utilitzant el paquet ar_markers_corrector més el script ekf, obtindrem els següents resultats.

10.2.2.1 Entorn real

Aquest entorn es tracta del circuit construït al CIRS. En aquest entorn, vaig realitzar diversos recorreguts de prova fins a registrar-ne un correctament. A continuació, es pot observar el recorregut real gravat per una càmera fixe, així com una foto del recorregut vist des de l'rviz per poder explicar-ne millor els resultats.

El vídeo del recorregut real el podeu veure aquí.

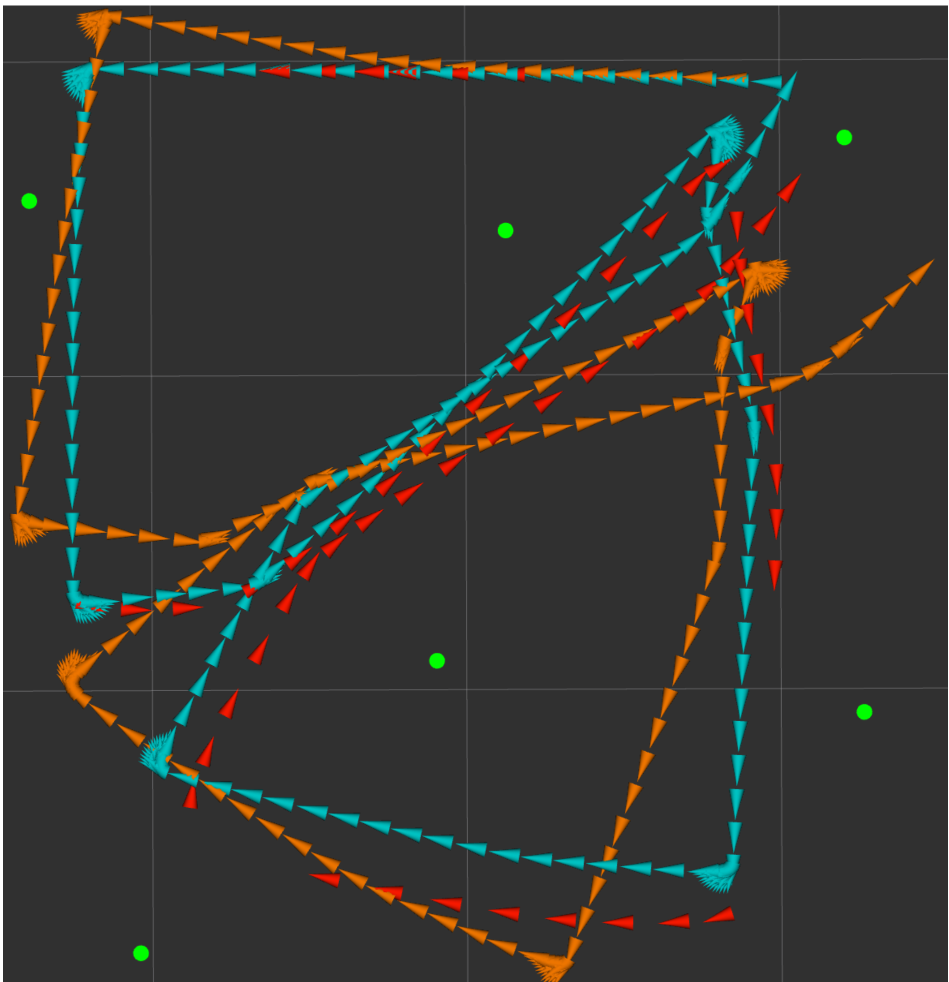
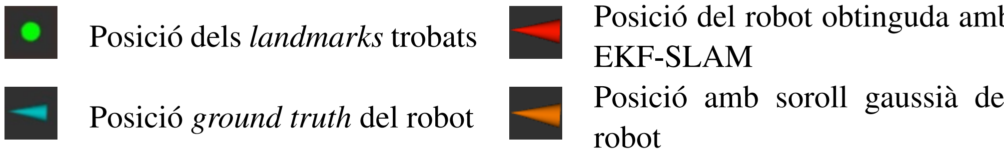


Figura 10.4: Recorregut real capturat amb l'rviz



Com es pot observar, la localització del robot és sovint correcte però presenta errors importants en alguns casos. Com es pot observar també, la trajectòria del robot no és del tot errònia, però presenta un defecte en l'orientació que pot donar-se per la inconsistència del sensor que utilitzem alhora de proporcionar la posició dels *markers*. Aquest error provoca que es col·loquin els *markers* incorrectament, fent creure al algoritme que es troba en un punt que no és i, per tant, afecta a la correcta localització del robot.

Per exemple, en aquest cas podem observar una bona localització:

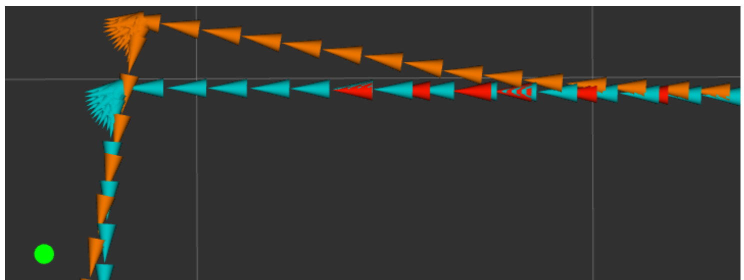
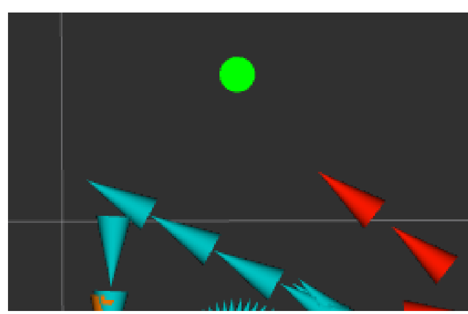


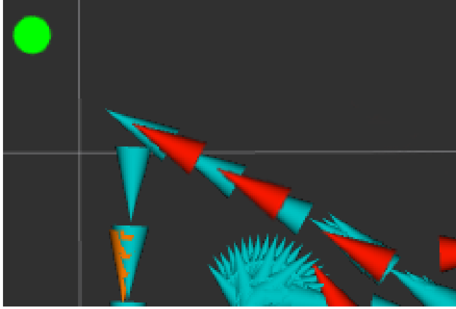
Figura 10.5: Detall de la figura 10.4

Aquest *marker* està ben col·locat i localitzat pel robot, això fa que la odomentria amb soroll quedi totalment filtrada per l'algoritme i sigui corregida correctament, casi a la perfecció, amb referència al *ground truth*.

Per altra banda, tenim aquest cas on la localització falla:



(a) Detall de la figura 10.4



(b) Detall de la figura 10.4 (corregit)

En aquest cas, com la correcció que he fet al sensor no es consistent, el *marker* queda localitzat erròniament i això afecta a la localització del nostre robot. Si el sistema hagués localitzat correctament el *marker*, el recorregut hauria estat correcte tal i com il·lustra la figura 10.6b, on he col·locat el marker a la seva posició de manera aproximada moguen conjuntament l'odomentria corregida.

10.2.2.2 Entorn simulat

Aquest entorn imita l'entorn real del CIRS amb unes petites variacions. En la següent imatge podem observar els resultats del entorn simulat vist des de l'rviz.

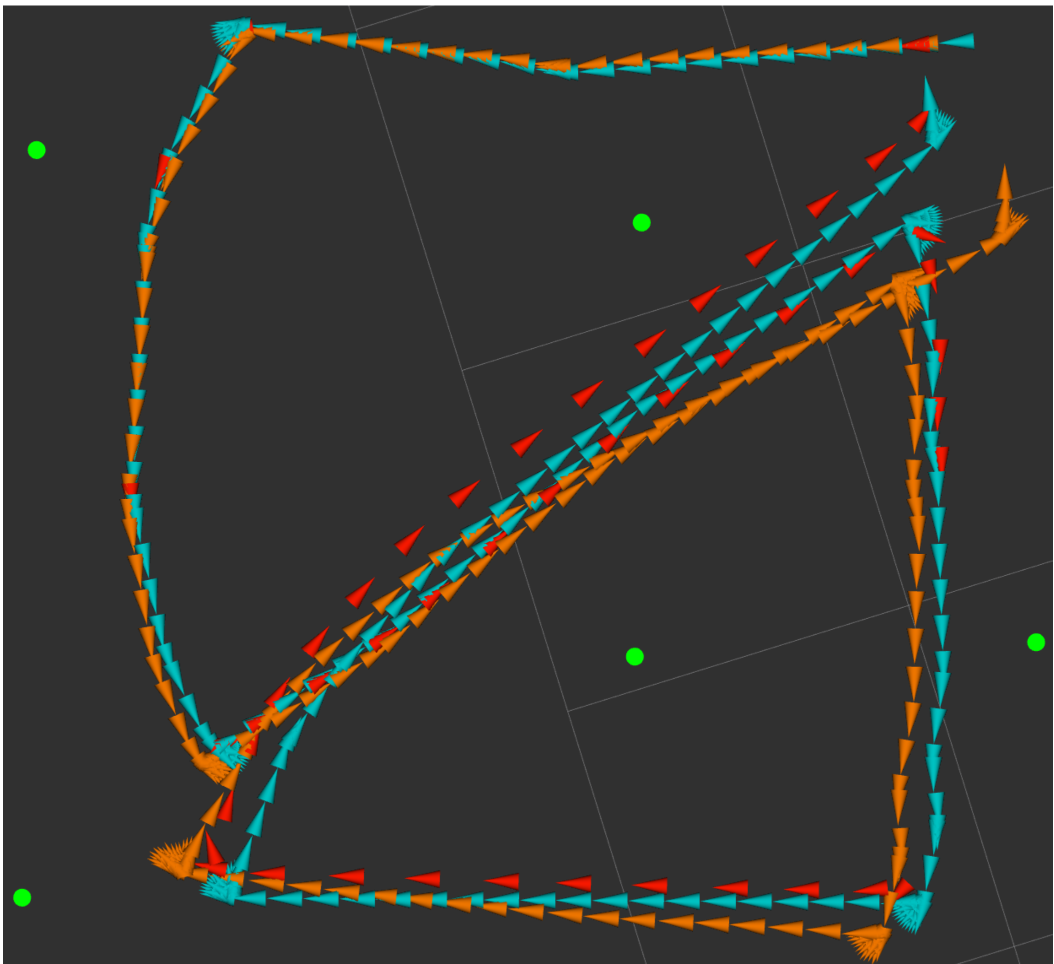
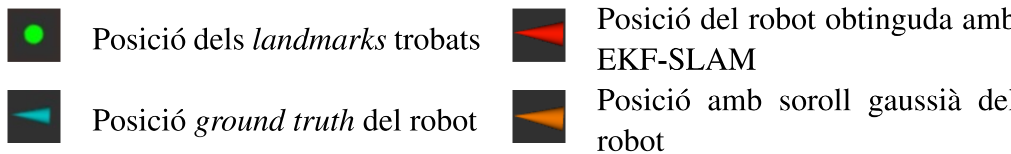


Figura 10.7: Dades finals d'incertesa



Com es pot observar en l'entorn simulat, l'algoritme implementat funciona molt millor filtrant el soroll de la odomentria. Aixo es deu a que en la simulació l'error inicial de la càmera comentat a la Secció 9.4 no existeix. Per aixo el sistema es molt millor localitzant el robot.

10.2.3 Pràctica de robòtica mòbil

Com a part d'aquest treball, aprofitant la renovació del Turtlebots i la implementació del algoritme EKF-SLAM, també hem preparat una pràctica per què els alumnes puguin desenvolupar el seu propi EKF-SLAM i aprendre en el procés. Amb això en ment, i utilitzant Jupyter notebook, hem fet una pràctica que explica les equacions clau del EKF, així com el seu funcionament.

La pràctica es pot veure a la Secció 13.5. També s'ha fet un fitxer similar amb la pràctica solucionada, però aquesta no pot ser adjuntada en aquest treball per motius obvis.

Capítol 11
Conclusions

L'objectiu principal d'aquest treball ha sigut realitzar un algoritme de localització que en un futur serveixi als alumnes per a realitzar pràctiques on ells mateixos es construeixen el seu propi algoritme i aprenguin els aspectes del EKF. Analitzant els resultats obtinguts i la redacció de la pràctica es pot considerar com a assoliment dels objectius inicials.

Com s'ha observat a l'apartat 10.2.2 l'algoritme és funcional i capaç de corregir l'error de l'odomentria utilitzant *markers* ArUco com a *landmarks*. També demostra que treballar amb entorns simulats, tot i que molt útil, pot suposar una gran diferència amb aquells problemes que et pots trobar al món real. Així doncs, és molt important realitzar proves reals, amb entorns que se semblin a allò que es trobarà habitualment el robot.

Durant tot el treball he pogut aprendre molts conceptes nous, així com consolidar aquells apresos durant els estudis. Tot i que no ha estat un camí fàcil, he aconseguit assolir els objectius proposats i poder finalitzar el treball. Espero, doncs, que la meva feina serveixi per a molts estudiants i que els motivi a endinsar-se en el món de la robòtica mòbil.

Capítol 12

Treball futur

Tot i haver complert tots el requisits inicials, sempre hi ha lloc per a millores dins el software.

Durant el calibratge de la càmera Intel RealSense, ens vam trobar que tot i seguir els passos del fabricant, teníem un gran error, sobretot en llargues distàncies. Aquest error el vam acabar corregint amb un pla corrector tal i com s'explica a la Subsubsecció 9.6.3.1. De totes maneres com hem vist a la Subsubsecció 10.2.2.1 aquesta correcció no sempre es suficient per localitzar el robor de manera precisa. Per tant, per millorar el programa, es podria investigar l'origen de l'error. Provar diferents mètodes de calibratge i comprovar quin dóna millors resultats, mirar que no es tracti d'un error de hardware, etc.

Pel que fa al codi, es podria millorar l'EKF utilitzant els *markers*, vistos de manera mes eficient. En aquest algoritme, els *markers* es tracten un per segon i s'afegeixen al filtre directament. La millora consistiria en tractar tots els makers i registrar-ne la seva entrada i només afegir-lo al filtre, sempre i quan es compleixin una sèrie de condicions. Després, abans d'afegir-lo, ja que tindríem una col·lecció de *markers*, podríem fer una mediana entre les distàncies per tal de millorar la precisió de les mesures.

Utilitzant el mateix robot i sensors, juntament amb l'algoritme de localització, podríem arribar a construir mapes 3D a mesura que moguéssim el robot. En aquest aspecte, hi ha molta feina a fer i, fins i tot, podria ser un nou TFG.

Un altre aspecte a millorar seria el hardware. Sempre queda espai, tal i com ha fet el meu company, i jo, d'afegir nous sensors al robot i, fins i tot, fer-los treballar conjuntament.

Capítol 13

Annexos

13.1 Script mesurador de distàncies ArUco

```
import csv
import os
import sys
import cv2
import rospy
import math
import statistics
from visualization_msgs.msg import MarkerArray

imageCounter = 0
cv_image = 0

def writeData():
 f = open(path + '.csv', 'w')
 f.write("xReal\tzReal\tx\t\tz\n")
 f.write("\t\tmean\tstdev\tmean\tstdev\n")
 text = path.replace('-',".").split('_')
 f.write(text[0] + "\t" + text[1] + "\t" +
  ↪  str(statistics.mean(markerInfo[0])) + "\t" +
  ↪  str(statistics.stdev(markerInfo[0])) + "\t" +
  ↪  str(statistics.mean(markerInfo[1])) + "\t" +
  ↪  str(statistics.stdev(markerInfo[1])) + "\n")

def callbackMarker(data):
    global imageCounter
    global markerInfo
    if  data.markers[0].id == 16:
     markerInfo[0].append(data.markers[0].pose.position.x)
     markerInfo[1].append(data.markers[0].pose.position.z)
     imageCounter += 1
     if imageCounter >= 100:
```

```
28            writeData()
29            os._exit(0)
30 
31    def getMarkerAruco():
32        rospy.init_node('listener', anonymous=True)
33        rospy.Subscriber('/ar_detector/ar_markers', MarkerArray,
           ↪  callbackMarker)
34        rospy.spin()
35 
36    args = sys.argv[1:]
37    if len(args) == 1:
38        path = args[0]
39        markerInfo = [[],[]]
40        getMarkerAruco()
41    else:
42        print("Incorrect args")
```

13.2 Paquet ar_marker_corrector

13.2.1 Corrector de markers

```
#!/usr/bin/env python3
import sys
import rospy
import time
import os
import tf2_ros
import tf2_geometry_msgs
from visualization_msgs.msg import MarkerArray
from ar_markers_correction.msg import PointArray
from ar_markers_correction.msg import BetterPoints

def getPoly23(p,x,y):
 return p[0] + p[1]*x + p[2]*y + p[3]*x**2 + p[4]*x*y +
  ↪ p[5]*y**2 + p[6]*x**2*y + p[7]*x*y**2 + p[8]*y**3

def getPoly2(p,x):
 return p[0]*x**2+p[1]*x+p[2]

def transformPoint(point,transform):
 return tf2_geometry_msgs.do_transform_point(point, transform)

def noTranslation(transform):
 transform.transform.translation.x = 0.0
 transform.transform.translation.y = 0.0
 transform.transform.translation.z = 0.0
 return transform

class MarkerCorrector(object):
 def __init__(self):
  self.polyX = [-0.01141, 1.036, 0.01859, 0.003187, -0.04202,
   ↪ -0.00806, -0.00182, 0.00473, 0.001254]
  self.polyZ = [0.01062, 0.0008652, 0.9775, -0.007451,
   ↪ -0.0006507, -0.01042, -0.003513, 0.0001064, 0.0003376]
  self.polyX_dev = [-0.0016,0.00828,0]
  self.polyZ_dev = [0.0009857,0.006637,0]
  self.markersAcceptats = [16,17,18,19,20,21,22,23]

```

```
35         self.tf_buffer = tf2_ros.Buffer(rospy.Duration(1200.0)) #tf
            ↪ buffer length
36         self.tf_listener = tf2_ros.TransformListener(self.tf_buffer)
37 
38         self.transform =
            ↪ self.tf_buffer.lookup_transform("base_link",
            ↪ "camera_aruco", rospy.Time(0), rospy.Duration(1.0))
39 
40         self.pub =
            ↪ rospy.Publisher('correctedPosition',PointArray,queue_size=10)
41         self.sub = rospy.Subscriber('/ar_detector/ar_markers',
            ↪ MarkerArray, self.callbackMarker)
42 
43     def markerAcceptat(self,marker):
44         return marker.id in self.markersAcceptats and
            ↪ getPoly23(self.polyZ,marker.pose.position.x,
            ↪ marker.pose.position.z) <= 4.5
45 
46     def callbackMarker(self,data):
47             markA = PointArray()
48             for marker in data.markers:
49              if self.markerAcceptat(marker):
50               markA.points.append(self.publishMarker(marker))
51             self.pub.publish(markA)
52 
53     def publishMarker(self,markerA):
54         marker = BetterPoints()
55 
56         marker.position.header = markerA.header
57         marker.position.header.frame_id="camera_aruco"
58 
59         marker.deviation.header = markerA.header
60         marker.deviation.header.frame_id="camera_aruco"
61 
62         marker.id = markerA.id
63 
64         marker.position.point.x =
            ↪ getPoly23(self.polyX,markerA.pose.position.x,
            ↪ markerA.pose.position.x)
65         marker.position.point.y = markerA.pose.position.y
```

```
66        marker.position.point.z =
           ↪  getPoly23(self.polyZ,markerA.pose.position.x,
           ↪  markerA.pose.position.z)
67
68        marker.deviation.point.x =
           ↪  getPoly2(self.polyX_dev,abs(marker.position.point.x))
69        marker.deviation.point.y = 0.01 #No utilitzat
70        marker.deviation.point.z =
           ↪  getPoly2(self.polyZ_dev,abs(marker.position.point.z))
71
72        marker.position =
           ↪  transformPoint(marker.position,self.transform)
73        marker.deviation = transformPoint(marker.deviation,
           ↪  noTranslation(self.transform))
74
75        return marker
76
77  if __name__ == '__main__':
78      rospy.init_node('ar_marker_corrector')
79      node = MarkerCorrector()
80      rospy.spin()
```

13.2.2 Generador de soroll per l'odometria

```
1  #!/usr/bin/env python3
2
3  import sys
4  import rospy
5  import numpy as np
6  import tf
7  from math import *
8
9  from nav_msgs.msg import Odometry
10
11 def wrap_angle(ang):
12         return ang + (2.0 * np.pi * np.floor((np.pi - ang) /
           ↪  (2.0 * np.pi)))
13
14 class ExtendedKalmanFilter:
15     "Generic Extended Kalman filter class"
16
17     def __init__(self, x, P, normalize_angle=None):
18         assert x.shape[0] == P.shape[0]
19         assert P.shape[0] == P.shape[1]
20
21         self.x = x
22         self.P = P
23         if normalize_angle is None:
24             self.normalize_angle = [False for _ in
               ↪  range(len(x))]
25         else:
26             assert len(normalize_angle) == x.shape[0]
27             self.normalize_angle = normalize_angle
28
29     def get_x(self):
30         return np.copy(self.x)
31
32     def get_P(self):
33         return np.copy(self.P)
34
35     def prediction(self, inc_t=0.0, u=None, Q=None):
36         self.x = self.calculate_f(inc_t, u)
37         A = self.calculate_A(inc_t, u)
```

```
38            W = self.calculate_W(inc_t, u)
39            self.P = A.dot(self.P.dot(A.transpose())) +
               ↪  W.dot(Q.dot(W.transpose()))
40
41        def calculate_f(self, inc_t=0.0, u=None):
42            """ implement in inherited class """
43            pass
44
45        def calculate_A(self, inc_t=0.0, u=None):
46            """ implement in inherited class """
47            pass
48
49        def calculate_B(self, inc_t=0.0, u=None):
50            """ implement in inherited class """
51            pass
52
53        def calculate_W(self, inc_t=0.0, u=None):
54            """ implement in inherited class """
55            pass
56
57    class VelocityIntegrator(ExtendedKalmanFilter):
58        "Class to integrate the Turtlebot velocities between two
           ↪  time steps"
59
60        def __init__(self):
61            ExtendedKalmanFilter.__init__(self, np.zeros(3),
               ↪  np.eye(3)*0.001)
62
63        def calculate_f(self, inc_t=0.0, u=None):
64            """ Computes f(x, u) """
65            return self.calculate_A().dot(self.x) +
               ↪  self.calculate_B(inc_t=inc_t).dot(u)
66
67        def calculate_A(self, inc_t=0.0, u=None):
68            """ implement in inherited class """
69            return np.eye(3)
70
71        def calculate_B(self, inc_t=0.0, u=None):
72            """ implement in inherited class """
73            ct = np.cos(self.x[2]) * inc_t
74            st = np.sin(self.x[2]) * inc_t
```

```
75            return np.array([ct, -st, 0, st, ct, 0, 0, 0,
               ↪  inc_t]).reshape(3, 3)
76
77        def calculate_W(self, inc_t=0.0, u=None):
78            """ implement in inherited class """
79            ct2 = np.cos(self.x[2]) * inc_t**2 / 2
80            st2 = np.sin(self.x[2]) * inc_t**2 / 2
81            return np.array([ct2, -st2, 0, st2, ct2, 0, 0, 0,
               ↪  inc_t**2/2]).reshape(3, 3)
82
83
84  class addNoiseOdom(object):
85   def __init__(self):
86    self.last_v_time = rospy.Time.now().to_sec()
87
88    self.integrator = VelocityIntegrator()
89
90    self.pub =
       ↪  rospy.Publisher('odomNoise',Odometry,queue_size=10)
91    self.sub = rospy.Subscriber('/odom', Odometry,
       ↪  self.addNoise)
92
93   def addNoise(self, odom):
94            t = rospy.Time.now().to_sec()
95            inc_t = t - self.last_v_time
96            if inc_t > 0.1:
97                inc_t = 0.1
98
99            x = np.array([odom.twist.twist.linear.x,
               ↪  odom.twist.twist.linear.y,
               ↪  odom.twist.twist.angular.z])
100
101           x[0] = np.random.normal(x[0],0.07,1)[0]
102           x[2] = np.random.normal(x[2],0.06,1)[0]
103
104           self.integrator.prediction(inc_t=inc_t, u=x,
               ↪  Q=np.eye(3))
105           xI = self.integrator.get_x()
106
107           odom.header.frame_id = "odom_noise"
108           odom.pose.pose.position.x = xI[0]
```

```
109            odom.pose.pose.position.y = xI[1]
110
111            quaternion =
               ↪  tf.transformations.quaternion_from_euler(0, 0,
               ↪  xI[2])
112
113            odom.pose.pose.orientation.x = quaternion[0]
114            odom.pose.pose.orientation.y = quaternion[1]
115            odom.pose.pose.orientation.z = quaternion[2]
116            odom.pose.pose.orientation.w = quaternion[3]
117
118            self.pub.publish(odom)
119            self.last_v_time = t
120
121    if __name__ == '__main__':
122        rospy.init_node('odomNoise')
123        node = addNoiseOdom()
124        rospy.spin()
```

13.3 EKF-SLAM

```
import numpy as np
import rospy
import tf
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Quaternion
from tf.transformations import euler_from_quaternion,
↪ quaternion_from_euler

from ar_markers_correction.msg import PointArray
from ar_markers_correction.msg import BetterPoints
from visualization_msgs.msg import Marker
from visualization_msgs.msg import MarkerArray

from multiprocessing import Process, Lock

def wrap_angle(ang):
    return ang + (2.0 * np.pi * np.floor((np.pi - ang) / (2.0
    ↪ * np.pi)))

def calculateDistance(z, Z):
    return

class ExtendedKalmanFilter:
    "Generic Extended Kalman filter class"

    def __init__(self, x, P, normalize_angle=None):
        assert x.shape[0] == P.shape[0]
        assert P.shape[0] == P.shape[1]

        self.x = x
        self.P = P
        if normalize_angle is None:
            self.normalize_angle = [False for _ in
            ↪ range(len(x))]
        else:
            assert len(normalize_angle) == x.shape[0]
            self.normalize_angle = normalize_angle

```

```
37      def get_x(self):
38          return np.copy(self.x)
39
40      def get_P(self):
41          return np.copy(self.P)
42
43      def prediction(self, inc_t=0.0, u=None, Q=None):
44          self.x =
45          A =
46          W =
47          self.P =
48
49      def update(self, y, R, s=None): # Linear Update
50          # Compute innovation
51
52          z =
53          H =
54          Z =
55          K =
56
57          if self.compatibilityTest(z,Z):
58           self.x =
59           self.P =
60
61      def calculate_f(self, inc_t=0.0, u=None):
62          """ implement in inherited class """
63          pass
64
65      def calculate_A(self, inc_t=0.0, u=None):
66          """ implement in inherited class """
67          pass
68
69      def calculate_B(self, inc_t=0.0, u=None):
70          """ implement in inherited class """
71          pass
72
73      def calculate_W(self, inc_t=0.0, u=None):
74          """ implement in inherited class """
75          pass
76
77      def calculate_h(self, s=None):
```

```
78            """ implement in inherited class """
79            pass
80
81        def calculate_H(self, s=None):
82            """ implement in inherited class """
83            pass
84
85        def compatibilityTest(self, z, Z):
86            return True
87
88    class TurtlebotVelocityEkf(ExtendedKalmanFilter):
89        "Class to estimate the Turtlebot linear and angular
           ↪  velocities"
90
91        def __init__(self, x, P):
92            ExtendedKalmanFilter.__init__(self, x, P)
93
94        def calculate_f(self, inc_t, u):
95            """ Computes f(x, u) """
96            return
97
98        def calculate_A(self, inc_t=0, u=None):
99            """ implement in inherited class """
100           return
101
102       def calculate_W(self, inc_t=0, u=None):
103           """ implement in inherited class """
104           return
105
106       def calculate_h(self, s=None):
107           """ implement in inherited class """
108           return
109
110       def calculate_H(self, s=None):
111           """ implement in inherited class """
112           return
113
114   class VelocityIntegrator(ExtendedKalmanFilter):
115       "Class to integrate the Turtlebot velocities between two
           ↪  time steps"
116
```

```
117     def __init__(self):
118         ExtendedKalmanFilter.__init__(self, np.zeros(3),
          ↪ np.eye(3)*0.001)
119 
120     def calculate_f(self, inc_t=0.0, u=None):
121         """ Computes f(x, u) """
122         return
123 
124     def calculate_A(self, inc_t=0.0, u=None):
125         """ implement in inherited class """
126         return
127 
128     def calculate_B(self, inc_t=0.0, u=None):
129         """ implement in inherited class """
130         return
131 
132     def calculate_W(self, inc_t=0.0, u=None):
133         """ implement in inherited class """
134         return
135 
136 class TurtlebotEkfSlam(ExtendedKalmanFilter):
137     "Class to estimate the Turtlebot linear and angular
      ↪ velocities using landmarks"
138 
139     def __init__(self, x, P):
140         ExtendedKalmanFilter.__init__(self, x, P)
141 
142     def compatibilityTest(self, z, Z):
143         return
144 
145     def calculate_f(self, inc_t=0.0, u=None):
146         """ Computes f(x, u) """
147         f =
148         return f
149 
150     def calculate_A(self, inc_t=0.0, u=None):
151         """ implement in inherited class """
152         A =
153         return A
154 
155     def calculate_W(self, inc_t=0.0, u=None):
```

```
156             """ implement in inherited class """
157             W =
158             return W
159
160         def calculate_h(self, s=None):
161             assert s >= 0 and s < (len(self.x) - 3) / 2
162             h=
163             return h
164
165         def calculate_H(self,s=None):
166             assert s >= 0 and s < (len(self.x) - 3) / 2
167             H =
168             return H
169
170
171         def state_augmentation(self, y_, R):
172
173             # Calculate g(x, y)
174             lxi =
175             lyi =
176
177             # Calculate GR
178             GR = i
179
180             # Calculate Gy
181             Gy =
182
183             # PLL & PLR
184             PLL =
185             PLR =
186
187             # Augment state
188             x_plus =
189
190             # Augment Covariance Matrix
191             P_plus =
192
193             self.x = x_plus
194             self.P = P_plus
195
196     class RosFilter:
```

```
197     def __init__(self):
198         self.velocity_filter =
             ↪  TurtlebotVelocityEkf(np.zeros(3), np.eye(3)*0.05)
199         self.integrator = VelocityIntegrator()
200         self.ekf_slam = None
201 
202         self.last_v_time = rospy.Time.now().to_sec()
203         self.last_marker_time = rospy.Time.now().to_sec()
204         self.last_odom = None
205         self.subs = rospy.Subscriber("/odomNoise", Odometry,
             ↪  self.callback)
206         self.subs2 =
             ↪  rospy.Subscriber("/correctedPosition",PointArray ,
             ↪  self.landmarkUpade)
207         self.num_landmarks = 0
208         self.markerId = {}
209         self.mutex = Lock()
210 
211 
212     def landmarkUpade(self, data):
213      # Tractar landmark
214 
215     def callback(self, odom):
216      # Filtar velocitat
217 
218 if __name__ == "__main__":
219     rospy.init_node('turtlebot_velocity_estimation',
         ↪  anonymous=True)
220     ekf = RosFilter()
221     rospy.spin()
```

13.4 Generador de models ArUco per gazebo

```
import cv2
import cv2.aruco as aruco
import argparse
import numpy as np
import os


class CreateAruco:
 def __init__(self,aruco_dict, marker_pixel, border_pixel,
  ↪ marker_size, path_to_save):
  self.marker_id = 0
  self.marker_image = None
  self.aruco_dict = aruco_dict
  self.marker_pixel = marker_pixel
  self.border_pixel = border_pixel
  self.marker_size = marker_size
  self.path_to_save = path_to_save
  self.make_dir(self.path_to_save)

 def make_dir(self,path):
  try:
   os.makedirs(path)
  except OSError:
   pass

 def create_and_save_a_marker(self,marker_id):
  self.marker_id = marker_id
  self.make_dir(self.path_to_save + "/tag_" +
   ↪ str(self.marker_id) + "/materials/textures")
  self.make_dir(self.path_to_save + "/tag_" +
   ↪ str(self.marker_id) + "/materials/scripts")
  self.create_marker()
  self.save_model_image()
  self.save_model_config()
  self.save_model_materials()
  self.save_model_sdf()
  print("create and save marker:",self.marker_id)

 def create_and_save_multi_marker(self,marker_id_list):
```

```
37      for marker_id in marker_id_list:
38       self.create_and_save_a_marker(marker_id)
39
40    def create_marker(self):
41      image = aruco.drawMarker(aruco_dict, self.marker_id,
         ↪  self.marker_pixel)
42       self.marker_image = cv2.copyMakeBorder(image,
          ↪  self.border_pixel, self.border_pixel,
          ↪  self.border_pixel, self.border_pixel,
          ↪  cv2.BORDER_CONSTANT, None, [255,255,255])
43
44    def save_model_image(self):
45      cv2.imwrite(self.path_to_save + "/tag_"+ \
         ↪  str(self.marker_id) +
         ↪  "/materials/textures/aruco_marker_"+ \
         ↪  str(self.marker_id) + ".png", self.marker_image)
46
47    def save_model_materials(self):
48      file = open(self.path_to_save + "/tag_" +
         ↪  str(self.marker_id) + "/materials/scripts/tag.material",
         ↪  'w')
49      file.write("\n \
50       material aruco_tag_"+str(self.marker_id)+"\n \
51        {\n \
52              technique\n \
53         {\n \
54               pass\n \
55               {\n \
56                texture_unit\n \
57                {\n \
58                 // Relative to the location of the material
    ↪   script\n \
59                 texture
    ↪   ../textures/aruco_marker_"+str(self.marker_id)+".png\n \
60                 // Repeat the texture over the surface (4 per
    ↪   face)\n \
61                 scale 1 1\n \
62             }\n \
63            }\n \
64         }\n \
65        }\n")
```

```
 file.close()

def save_model_config(self):
 file = open(self.path_to_save + "/tag_" + str(self.marker_id) + "/model.config", 'w')
 file.write("\n \
  <?xml version=\"1.0\"?>\n \
  \n \
  <model>\n \
  <name>Aruco tag"+ str(self.marker_id) +"</name>\n \
  <version>1.0</version>\n \
  <sdf version=\"1.6\">model.sdf</sdf>\n \
  \n \
  <author>\n \
  <name>Nievas Martin</name>\n \
  <email>martin.nievas.ar@gmail.com</email>\n \
  </author>\n \
  \n \
  <description>\n \
  Aruco tag "+str(self.marker_id)+"\n \
  </description>\n \
  \n \
  </model>\n \")
 file.close()


def save_model_sdf(self):
 file = open(self.path_to_save + "/tag_" + str(self.marker_id) + "/model.sdf", 'w')
 file.write("\n \
  <?xml version=\"1.0\"?>\n \
   <sdf version=\"1.6\">\n \
    <model name=\"Aruco tag"+ str(self.marker_id) + "\">\n \
     <static>true</static>\n \
      <link name=\"robot_link\">\n \
       <visual name=\"body_visual\">\n \
        <geometry>\n \
         <box>\n \
          <size>" + str(marker_size[0]) + " " + str(marker_size[1]) + " " + str(marker_size[2]) +" " + " </size>\n \
```

```
103             </box>\n \
104                    </geometry>\n \
105                   <material> <!-- Body material -->\n \
106                <script>\n \
107                 <uri>model://tag_" + str(self.marker_id) +
    ↪  "/materials/scripts/tag.material</uri>\n \
108                     <name>aruco_tag_" + str(self.marker_id) +
    ↪  "</name>\n \
109                    </script>\n \
110                   </material> <!-- End Body Material -->\n \
111              </visual>\n \
112             </link>\n \
113            </model>\n \
114           </sdf>\n \ \n")
115    file.close()
116
117
118 if __name__ == "__main__":
119   aruco_dict = aruco.Dictionary_get(16)
120   marker_pixel = 900
121   marker_size = [0.15,0.15,0.015]
122   border_pixel = 50
123   path_to_save = "./models"
124   aruco_gazebo = CreateAruco(aruco_dict, marker_pixel,
     ↪  border_pixel, marker_size, path_to_save)
125   for i in range(10):
126    aruco_gazebo.create_and_save_a_marker(i+20)
```

13.5 Pràctica de robòtica mòbil

Turtlebot motion filter

Turtlebot publishes an `/odom` topic of type `nav_msgs/Odometry`. This message contains the velocity estimated by the robot as well as its position computed integrating these velocities over time. What we want to do here is:

- Filter out the forward and angular velocities (i.e., v and w) using a constant velocity model.
- Estimate the robot pose increment (i.e., Δ_x, Δ_y, and Δ_ψ) and its uncertainty for a specific period of time. To do it we are going to use a Kalman filter and an integrator.

Extended Kalman filter

Despite filtering a velocity using a constant velocity model is a lineal task, lets implement an Extended Kalman filter that is the more general form and it will allow us to reuse it in the future.

The equations to implement it are the following:

Prediction:

$$\hat{\mathbf{x}} = f(\hat{\mathbf{x}}, \mathbf{u})$$

$$\mathbf{P} = \mathbf{A}\mathbf{P}\mathbf{A}^T + \mathbf{W}\mathbf{Q}\mathbf{W}^T$$

Update:

$$\hat{\mathbf{z}} = \mathbf{y} - h(\hat{\mathbf{x}})$$

$$\mathbf{Z} = \mathbf{H}\mathbf{P}\mathbf{H}^T + \mathbf{R}$$

$$\mathbf{K} = \mathbf{P}\mathbf{H}^T\mathbf{Z}^{-1}$$

$$\hat{\mathbf{x}} = \hat{\mathbf{x}} + \mathbf{K}\hat{\mathbf{z}}$$

$$\mathbf{P} = \mathbf{P} - \mathbf{K}\mathbf{Z}\mathbf{K}^T$$

In [2]:

```
import numpy as np

class ExtendedKalmanFilter:
    "Generic Extended Kalman filter class"

    def __init__(self, x, P, normalize_angle=None):
        assert x.shape[0] == P.shape[0]
        assert P.shape[0] == P.shape[1]

        self.x = x
        self.P = P
        if normalize_angle is None:
            self.normalize_angle = [False for _ in range(len(x))]
        else:
            assert len(normalize_angle) == x.shape[0]
            self.normalize_angle = normalize_angle

    def get_x(self):
        return np.copy(self.x)

    def get_P(self):
        return np.copy(self.P)

    def prediction(self, inc_t, u, Q):
        self.x = self.calculate_f(inc_t, u)
        A = self.calculate_A(inc_t)
        W = self.calculate_W(inc_t)
        self.P = A.dot(self.P.dot(A.transpose())) + W.dot(Q.dot(W.transpose()))

    def update(self, y, R, s=None): # Linear Update
        # Compute innovation
        z = y - self.calculate_h(s)
        H = self.calculate_H(s)
        Z = H.dot(self.P.dot(H.transpose())) + R
        K = self.P.dot(H.transpose().dot(np.linalg.inv(Z)))
        if self.compatibilityTest(z,Z):
         self.x = self.x + K.dot(z)
         self.P = self.P - K.dot(Z.dot(K.transpose()))

    def calculate_f(self, inc_t, u):
        """ implement in inherited class """
```

```python
        pass

    def calculate_A(self, inc_t):
        """ implement in inherited class """
        pass

    def calculate_B(self, inc_t):
        """ implement in inherited class """
        pass

    def calculate_W(self, inc_t):
        """ implement in inherited class """
        pass

    def calculate_h(self, s):
        """ implement in inherited class """
        pass

    def calculate_H(self, s):
        """ implement in inherited class """
        pass

    def compatibilityTest(self, z, Z):
        return True
```

Turtlebot velocity estimation using a Kalman filter

Lets see the main equations involved in this task.

state

The state that we want to filter includes the turtlebot velocities:

$$\hat{\mathbf{x}} = [v_x \; v_y \; w]$$

motion model

A constant velocity model is going to be used:

$$f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n}) = \begin{cases} v_x = v_x + n_{vx} t \\ v_y = v_y + n_{vy} t \\ w = w + n_w t \end{cases}$$

where $\hat{\mathbf{x}}$ is the state, $\mathbf{u}$ is the model input and $\mathbf{n}$ is the noise. Here to simplify the filter the input $\mathbf{u}$ is empty. Normally it could contain the setpoints sent to the motor. The noise $\mathbf{n} = [n_{vx} \; n_{vy} \; n_w]$ is modelled as an acceleration, therefore, it is multiplied by the increment of time (t).

direct observation model

From the topic `/odom/twist` we observe the lineal and angular velocities ($\mathbf{y} = [y_{vx} \; y_{vy} \; y_w]$) that is exactly what we have in the state vector. Then, the observation model is lineal:

$$\mathbf{y} = h(\hat{\mathbf{x}}) + \mathbf{v} = \begin{cases} y_{vx} = v_x + \sigma_{vx} \\ y_{vy} = v_y + \sigma_{vy} \\ y_w = w + \sigma_w \end{cases}$$

being $\mathbf{v} = [\sigma_{vx} \; \sigma_{vy} \; \sigma_w]$ the noise in the sensor measurements.

Implementation

When the Kalman filter is lineal the motion model average can be computed using matrices $\mathbf{A}$ and $\mathbf{B}$.

$$f(\hat{\mathbf{x}}, \mathbf{u}) = \mathbf{A}\hat{\mathbf{x}} + \mathbf{B}\mathbf{u}$$

$$\mathbf{A} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n})}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To estimate the uncertainty in the state vector matrices $\mathbf{W}$ and $\mathbf{Q}$ are needed.

$$\mathbf{W} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n})}{\partial \mathbf{n}} = \begin{bmatrix} t & 0 & 0 \\ 0 & t & 0 \\ 0 & 0 & t \end{bmatrix}$$

$$\mathbf{Q} = \begin{bmatrix} n_{vx} & 0 & 0 \\ 0 & n_{vy} & 0 \\ 0 & 0 & n_w \end{bmatrix}$$

For the observation model, it can be said that if updates are lineal then:

$$h(\hat{\mathbf{x}}) = \mathbf{H}\hat{\mathbf{x}}$$

Therefore, matrices $\mathbf{H}$ and $\mathbf{R}$ have to be defined.

$$\mathbf{H} = \frac{\partial h(\hat{\mathbf{x}})}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} \sigma_{vx} & 0 & 0 \\ 0 & \sigma_{vy} & 0 \\ 0 & 0 & \sigma_w \end{bmatrix}$$

In [3]:

```
class TurtlebotVelocityEkf(ExtendedKalmanFilter):
    "Class to estimate the Turtlebot linear and angular velocities"

    def __init__(self, x, P):
        ExtendedKalmanFilter.__init__(self, x, P)

    def calculate_f(self, inc_t, u):
        """ Computes f(x, u) """
        return self.calculate_A(inc_t).dot(self.x)

    def calculate_A(self, inc_t):
        """ implement in inherited class """
        return np.eye(3)

    def calculate_W(self, inc_t):
        """ implement in inherited class """
        return np.eye(3)*inc_t

    def calculate_h(self, s):
        """ implement in inherited class """
        return self.calculate_H(s).dot(self.x)

    def calculate_H(self, s):
        """ implement in inherited class """
        return np.eye(3)
```

Integrate Position

Once we have the turtlebot velocity estimated with its uncertainty we can integrate it to compute the Δ_x, Δ_y, Δ_ψ between two time steps. To do it we can use the **prediction** step of a Kalman filter in which:

state

The state contains the position and orientation increments.

$$\hat{\mathbf{x}} = [\Delta_x \; \Delta_y \; \Delta_\psi]$$

input

The input contains the linear and angular velocity estimated by the previous filter.

$$\mathbf{u} = [v_x \; v_y \; w]$$

noise

The noise of the input is already estimated in the covariance matrix $\mathbf{P}$ in the previous filter.

$$\mathbf{n} = [n_{vx} \; n_{vy} \; n_w]$$

motion model

The motion model follows this equation:

$$f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n}) = \begin{cases} \Delta_x = \Delta_x + \cos(\Delta_\psi)(tv_x + \frac{t^2}{2}n_{vx}) - \sin(\Delta_\psi)(tv_y + \frac{t^2}{2}n_{vy}) \\ \Delta_y = \Delta_y + \sin(\Delta_\psi)(tv_x + \frac{t^2}{2}n_{vx}) + \sin(\Delta_\psi)(tv_y + \frac{t^2}{2}n_{v_y}) \\ \Delta_\psi = \Delta_\psi + tw + \frac{t^2}{2}n_w \end{cases}$$

Implementation

To implement this position integrator, the following matrices have to be defined:

$$\mathbf{A} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n})}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{B} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n})}{\partial \mathbf{u}} = \begin{bmatrix} \cos(\Delta_\psi t & -\sin(\Delta_\psi)t & 0 \\ \sin(\Delta_\psi t & \cos(\Delta_\psi)t & 0 \\ 0 & 0 & t \end{bmatrix}$$

$$\mathbf{W} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n})}{\partial \mathbf{n}} = \begin{bmatrix} \cos(\Delta_\psi \frac{t^2}{2} & -\sin(\Delta_\psi)\frac{t^2}{2} & 0 \\ \sin(\Delta_\psi \frac{t^2}{2} & \cos(\Delta_\psi)\frac{t^2}{2} & 0 \\ 0 & 0 & \frac{t^2}{2} \end{bmatrix}$$

The input $\mathbf{u}$ is the state $\hat{\mathbf{x}}$ from the velocity filter and $\mathbf{Q}$ is the diagonal of the covariance matrix $\mathbf{P}$ also from the velocity filter.

In [0]:

```
class VelocityIntegrator(ExtendedKalmanFilter):
    "Class to integrate the Turtlebot velocities between two time steps"

    def __init__(self):
        ExtendedKalmanFilter.__init__(self, np.zeros(3), np.zeros((3, 3)))

    def calculate_f(self, inc_t, u):
        """ Computes f(x, u) """
        return self.calculate_A(inc_t).dot(self.x) + self.calculate_B(inc_t).dot(u)

    def calculate_A(self, inc_t):
        """ implement in inherited class """
        return np.eye(3)

    def calculate_B(self, inc_t):
        """ implement in inherited class """
        ct = np.cos(self.x[2]) * inc_t
        st = np.sin(self.x[2]) * inc_t
        return np.array([ct, -st, 0, st, ct, 0, 0, 0, inc_t]).reshape(3, 3)

    def calculate_W(self, inc_t):
        """ implement in inherited class """
        ct2 = np.cos(self.x[2]) * inc_t**2 / 2
        st2 = np.sin(self.x[2]) * inc_t**2 / 2
        return np.array([ct2, -st2, 0, st2, ct2, 0, 0, 0, inc_t**2/2]).reshape(3, 3)
```

Turtlebot EKF SLAM

Here we want to combine the position increments $[\Delta_x \; \Delta_y \; \Delta_\psi]$ obtained from the `VelocityIntegrator` as inputs $\mathbf{u}$ in a EKF SLAM filter. While this increments will be used in the *motion model*, observations done with another sensor can be used to *update* the vehicle state.

state

$$\hat{\mathbf{x}} = [x \; y \; \psi \; l_{xi} \; l_{yi} \; \ldots \; l_{xn} \; l_{yn}]$$

The state is composed by $[x \; y \; \psi]$, the robot position and orientation with respect the $\{I\}$, and $[l_{xi} \; l_{yi}]$, the position of the landmark i also respect to $\{I\}$. Notice that, initially, the state $\hat{\mathbf{x}}$ contains only the robot position. Landmark positions are added to the state once they are observed for the first time. Therefore, the state can contain from 0 to n landmarks.

input

$$\mathbf{u} = [\Delta_x \; \Delta_y \; \Delta_\psi]$$

The input of the turtlebot EKF SLAM is the state $\hat{\mathbf{x}}$ obtained from the previous `VelocityIntegrator` module.

noise

$$\mathbf{n} = [n_x \; n_y \; n_\psi]$$

The noise in the motion model that affects the robot position correspond to the uncertainty estimating the increments Δ_x, Δ_y and Δ_ψ. This noise is estimated in the diagonal of the covariance matrix $\mathbf{P}$ computed in the previous `VelocityIntegrator` module.

When a landmark is detected by the robot, a noise has to be also associated to it.

$$\mathbf{n} = [n_{lxi} \; n_{lyi}]$$

This noise will allow us to *move* the position of the landmarks once initialized, and therefore, to improve it.

motion model

$$f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n}) = \begin{cases} x = x + \cos(\psi)(\Delta_x + n_x) - \sin(\psi)(\Delta_y + n_y) \\ y = y + \sin(\psi)(\Delta_x + n_x) + \cos(\psi)(\Delta_y + n_y) \\ \psi = \psi + (\Delta_\psi + n_\psi) \\ \vdots \\ l_{xn} = l_{xn} + n_{lxn} \\ l_{yn} = l_{yn} + n_{lyn} \end{cases}$$

update

To implement the update step, the observation model $h(\hat{\mathbf{x}})$ must be defined. Given that an observation is defened as $\mathbf{y} = [{}^v l_{ix}, {}^v l_{iy}]$ where ${}^v l_{ix}$ is the landmark i position in x with respect to the vehicle (v) (the same for ${}^v l_{iy}$). Then the function that relates this observation $\mathbf{y}$ with the state vector $\hat{\mathbf{x}}$ is defined as:

$$h(\hat{\mathbf{x}}) = \begin{cases} {}^v l_{ix} = -\cos(\psi)x - \sin(\psi)y + \cos(\psi)l_{ix} + \sin(\psi)l_{iy} \\ {}^v l_{iy} = \sin(\psi)x - \cos(\psi)y - \sin(\psi)l_{ix} + \cos(\psi)l_{iy} \end{cases}$$

where (l_{ix}, l_{iy}) is the position of landmark i with respect to the *world* frame and is what we store in $\hat{\mathbf{x}}$.

state augmentation

Every time that a new landmark is observerd, it must be added into the state vector.

$$\hat{\mathbf{x}}^+ = \begin{bmatrix} \hat{\mathbf{x}} \\ g(\hat{\mathbf{x}}, \mathbf{y}) \end{bmatrix}$$

being $g(\hat{\mathbf{x}}, \mathbf{y})$ the inverse of the observation model.

$$g(\hat{\mathbf{x}}, \mathbf{y}) = \begin{cases} l_{ix} = x + \cos(\psi){}^v l_{ix} - \sin(\psi){}^v l_{iy} \\ l_{iy} = y + \sin(\psi){}^v l_{ix} + \cos(\psi){}^v l_{iy} \end{cases}$$

compatibility test

Usually, when calculating distances between two points we use the euclidian distance but when these points have an uncertainty associated, you should use the Mahalanobis distance. This distance can be used to discard points that may afect negatively our update.

$$M^2 > z^T Z^{-1} z$$

To define our M^2 we must use the χ^2 distribution table. This table is definded by two values: the degrees of fredoom and the probability of a larger value of χ^2.

Implementation

Here the motion model is not linear and therefore $f(\hat{\mathbf{x}}, \mathbf{u}) \neq \mathbf{A}\hat{\mathbf{x}} + \mathbf{B}\mathbf{u}$. Then, to implement the EKF SLAM filter, both the $f(\hat{\mathbf{x}}, \mathbf{u})$ function and the following matrices have to be calculated:

$$\mathbf{A} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u})}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} 1 & 0 & -\sin(\psi)\Delta_x - \cos(\psi)\Delta_y & \dots & 0 & 0 \\ 0 & 1 & \cos(\psi)\Delta_x - \sin(\psi)\Delta_y & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 1 & 1 \end{bmatrix}$$

$$\mathbf{W} = \frac{\partial f(\hat{\mathbf{x}}, \mathbf{u}, \mathbf{n})}{\partial \mathbf{n}} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 & \dots & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

$$\mathbf{H} = \frac{\partial h(\hat{\mathbf{x}})}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} -\cos(\psi) & -\sin(\psi) & \sin(\psi)x - \cos(\psi)y - \sin(\psi)l_{ix} + \cos(\psi)l_{iy} & 0 \dots & \cos(\psi) & \sin(\psi) & \dots 0 \\ \sin(\psi) & -\cos(\psi) & \cos(\psi)x + \sin(\psi)y - \cos(\psi)l_{ix} - \sin(\psi)l_{iy} & 0 \dots & -\sin(\psi) & \cos(\psi) & \dots 0 \end{bmatrix}$$

Note that $\hat{\mathbf{x}}$ may contain n landmarks and therefore the last part of the $\mathbf{H}$ matrix must coincide with the correct one setting the rest to 0.

In the **state augmentation** step, to augment the covariance matrix $\mathbf{P}$, the partial derivatives of $g(\hat{\mathbf{x}}, \mathbf{y})$ with respect to the part of the state vector that refers to the robot (i.e., the first 3 elements $\hat{\mathbf{x}}[\mathbf{0}:\mathbf{3}]$) and with respect to $\mathbf{y}$ have to be calculated:

$$\mathbf{G_R} = \frac{\partial g(\hat{\mathbf{x}}, \mathbf{y})}{\partial \hat{\mathbf{x}}[\mathbf{0}:\mathbf{3}]} = \begin{bmatrix} 1 & 0 & -\sin(\psi)^v l_{ix} - \cos(\psi)^v l_{iy} \\ 0 & 1 & \cos(\psi)^v l_{ix} - \sin(\psi)^v l_{iy} \end{bmatrix}$$

$$\mathbf{G_y} = \frac{\partial g(\hat{\mathbf{x}}, \mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix}$$

The matrix $\mathbf{P}$ must be augmentated as

$$\mathbf{P}^+ = \begin{bmatrix} \mathbf{P} & \mathbf{P_{LR}}^T \\ \mathbf{P_L R} & \mathbf{P_{LL}} \end{bmatrix}$$

being

$$\mathbf{P_{LL}} = \mathbf{G_R P}[\mathbf{0}:\mathbf{3}, \mathbf{0}:\mathbf{3}]\mathbf{G_R}^T + \mathbf{G_y R G_y}^T$$
$$\mathbf{P_{LR}} = \mathbf{G_R P}[0:3, :]$$

ROS script

The following ROS script creates a subscriber to the `/odom` topic and estimates the vehicle linear and angular velocities. and

In [6]:

```
import numpy as np
a = np.array([1,2,3])
b = np.array([5,6,7])
c = np.hstack((a,b))
print(c)
print(c[3:])
```

```
[1 2 3 5 6 7]
[5 6 7]
```

In [0]:

```
import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist

class RosFilter:
    def __init__(self):
        self.ekf = TurtlebotVelocityEkf(np.zeros(3),
                                        np.eye(3)*0.01)

        self.last_time = rospy.Time.now().to_sec()
        self.pub = rospy.Publisher("/twist", Twist)
        self.subs = rospy.Subscriber("/odom", Odometry, self.callback)

    def callback(self, odom):
        t = rospy.Time.now().to_sec()
        self.ekf.prediction(self.last_time - t, None, np.eye(3)*0.02)
        self.ekf.update(np.array([odom.twist.twist.linear.x,
                                  odom.twist.twist.angular.z]),
                        np.eye(3)*0.001)

        self.last_time = t
        x = self.ekf.get_x()
        twist = Twist()
        twist.linear.x = x[0]
        twist.linear.y = x[1]
        twist.angular.z = x[2]
        self.pub.publish(twist)
```

Bibliografia

[ArU 021] *Grupo de investigación 'Aplicaciones de la Visión Artificial' (A.V.A) de la Universidad de Córdoba*. https://www.uco.es/investiga/grupos/ava/, (S'ha accedit: març 2021). (Cited on pages 3, 7 and 35.)

[Construct 2021] The Construct. *A Platform to Become a ROS Developer from Zero*. https://www.youtube.com/c/TheConstruct/featured, 2021. (Cited on pages 41 and 45.)

[Courses 2006] E. Courses and T Surveys. *Sigma-Point Filters: An Overview with Applications to Integrated Navigation and Vision Assisted Control. Nonlinear Statistical Signal Processing Workshop*, 2006. (Cited on page 8.)

[Gaz 2021] *Gazebo Tutorials*. http://gazebosim.org/tutorials, 2021. (Cited on page 41.)

[Hea 2020] *std_msgs/Header Message*. http://docs.ros.org/en/noetic/api/std_msgs/html/msg/Header.html, 2020. (Cited on page 20.)

[Int 021a] *Instal·lar calibrador Intel RealSense (PDF)*. https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense_D400_Dyn_Calib_User_Guide.pdf, (S'ha accedit: Febrer 2021). (Cited on pages 31 and 32.)

[Int 021b] *Intel® RealSense™ Depth Camera D435i datasheet*. https://www.intelrealsense.com/wp-content/uploads/2020/06/Intel-RealSense-D400-Series-Datasheet-June-2020.pdf, (S'ha accedit: Febrer 2021). (Cited on page 17.)

[Int 021c] *Llibreries ROS Intel RealSense*. https://github.com/IntelRealSense/librealsense/blob/development/doc/distribution_linux.md, (S'ha accedit: Febrer 2021). (Cited on page 32.)

[Int 021d] *Web oficial Intel® RealSense™ Depth Camera D435i*. https://www.intelrealsense.com/depth-camera-d435i/, (S'ha accedit: Febrer 2021). (Cited on page 17.)

[Kalman 1960] R.E. Kalman. *Contributions to the Theory of Optimal Control*, 1960. (Cited on pages 3 and 8.)

[Mar 2021a] *visualization_msgs/Marker Message*. http://docs.ros.org/en/noetic/api/visualization_msgs/html/msg/Marker.html, 2021. (Cited on page 19.)

[Mar 2021b] *visualization_msgs/MarkerArray.msg*. https://docs.ros.org/en/api/visualization_msgs/html/msg/MarkerArray.html, 2021. (Cited on page 18.)

[Nievas 021] Martin Nievas. *Gazebo aruco tag generator*, (S'ha accedit: juny 2021). Disponible a https://github.com/MartinNievas/gazebo_aruco_tag_generator. (Cited on pages 41 and 42.)

[Poi 2021a] *geometry_msgs/Point Message*. http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Point.html, 2021. (Cited on pages 20 and 21.)

[Poi 2021b] *geometry_msgs/PointStamped.msg*. http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/PointStamped.html, 2021. (Cited on page 21.)

[Pos 2021] *geometry_msgs/Pose Message*. http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Pose.html, 2021. (Cited on page 20.)

[ROS 021] *Creating a ROS Package*. http://wiki.ros.org/ROS/Tutorials/CreatingPackage, (S'ha accedit: abril 2021). (Cited on page 45.)

[Solà 2014] Joan Solà. Simultaneous localization and mapping with extended Kalman filter. 2014. (Cited on pages 24 and 52.)

[spa 2018] *Waiting for service /gazebo/spawn_urdf_model*. https://forum.humanbrainproject.eu/t/waiting-for-service-gazebo-spawn-urdf-model/468, 2018. (Cited on page 41.)

[URD 2020] *Using a URDF in Gazebo*. http://wiki.ros.org/urdf/Tutorials/Using%20a%20URDF%20in%20Gazebo, 2020. (Cited on page 41.)