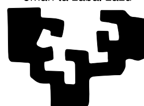
Mainbot robotaren simulazioa eta nabigazioa

Karrera Bukaerako Proiektua

2013.eko azaroaren 28

Aritz Arandia Obineta

Zuzendaria:
Elena Lazkano



Universidad del País Vasco
Euskal Herriko Unibertsitatea

Gaien Aurkibidea

1 Sarrera **1**
- 1.1 Sarrera 2
- 1.2 Proiektuaren deskribapen orokorra 2

2 Proiektuaren helburuen dokumentua **5**
- 2.1 Proiektuaren deskribapena eta helburuak 6
- 2.2 Proiektuaren abiapuntua 7
- 2.3 Ataza eta azpiatazen zerrenda 7
- 2.4 Lanaren Deskonposaketa Egitura (LDE diagrama) 8
- 2.5 Proiektuaren plangintza 8
  - 2.5.1 Estimazioa 10
- 2.6 Lan metodologia 11
- 2.7 Arriskuen analisia 12
  - 2.7.1 Arriskuen zerrenda 12
  - 2.7.2 Arriskuen kontingentzia plana 12
- 2.8 Denbora erreala 13

3 ROS **15**
- 3.1 Sarrera 17
  - 3.1.1 Ezaugarriak 17
- 3.2 Kontzeptu orokorrak 20
- 3.3 Fitxategi-sistema 20
  - 3.3.1 Paketeak 21
  - 3.3.2 Manifestuak 21
  - 3.3.3 Pilak (*Stack*) 22
  - 3.3.4 Pilen manifestuak (*Stack Manifest*) 22
  - 3.3.5 Mezu-mota (msg) 23
  - 3.3.6 Zerbitzu mota (srv) 23
  - 3.3.7 Launch fitxategiak 23
- 3.4 Konputazio maila 24
  - 3.4.1 Nodoak 24

3.4.2 ROS gainbegiralea . . . 24
3.4.3 Parametroen zerbitzaria . . . 25
3.4.4 Mezuak . . . 27
3.4.5 Topicak . . . 27
3.4.6 Zerbitzuak . . . 28
3.5 Goi mailako kontzeptuak . . . 28
3.5.1 Erreferentzia-sistemak/Transformatuak . . . 29
3.5.2 Robot eredua (URDF) . . . 31
3.6 Simulazioa . . . 33
3.6.1 Gazebo . . . 34

4 Ingurunea 37
4.1 Sarrera . . . 38
4.2 Ingurunearen URDF eredua . . . 38

5 Mainbot robotaren simulazioa 43
5.1 Mainbot robotaren deskribapena . . . 44
5.1.1 Mainbot robot erreala . . . 44
5.1.2 Robotaren URDF eredua . . . 47
5.1.3 Pluginak . . . 49
5.2 ROSeko pr2_mechanism pila . . . 53
5.2.1 Pr2_controller_interface paketea . . . 54
5.2.2 Pr2_controller_manager paketea . . . 56
5.2.3 Pr2_mechanism_model paketea . . . 56
5.3 Robotaren kontroladorea . . . 59
5.3.1 Ackermann mugimendu-sistema . . . 59
5.3.2 PID kontrola . . . 60
5.3.3 Kontroladorea idazten . . . 63
5.3.4 Gazebo Controller Manager plugina . . . 68
5.4 Exekuzioa . . . 68

6 Nabigazioa 71
6.1 Sarrera . . . 73
6.2 Odometria . . . 73
6.3 Nabigazioaren mapa . . . 75
6.3.1 Irudi-fitxategia . . . 75
6.3.2 YAML fitxategia . . . 76
6.4 Move_base paketea . . . 77
6.4.1 Berreskuratze-portaerak (*Recovery behaviors*) . . . 77
6.5 Kostu-mapak . . . 79
6.5.1 Sarrera . . . 79

6.5.2 Azalpen orokorra . . . 80
6.5.3 Markaketa eta garbiketa . . . 80
6.5.4 Espazio okupatua, librea eta ezezaguna . . . 80
6.5.5 tf . . . 80
6.5.6 *Puztea* . . . 81
6.5.7 Mapa motak . . . 81
6.5.8 Gure bi kostu-mapen arteko parametro komunak . . . 83
6.5.9 Kostu-mapa globala . . . 84
6.5.10 Kostu-mapa lokala . . . 84
6.6 Planifikatzaile globala . . . 84
6.7 Planifikatzaile lokala . . . 85
6.7.1 Gelaxka-sare mapa . . . 86
6.7.2 Gure konfigurazioa . . . 86
6.8 Nabigazio-sistemaren exekuzioa . . . 87
6.8.1 Launch fitxategia . . . 87
6.8.2 RViz tresna . . . 88
6.8.3 Nabigazio komandoak bidaltzea . . . 89
6.9 Exekuzioa . . . 90

7 Ondorioak eta etorkizunerako lana **95**
7.1 Ondorioak . . . 96
7.1.1 Ondorio pertsonalak . . . 97
7.2 Etorkizuneko lana . . . 97

Irudien Zerrenda

2.1 Lanaren Deskonposaketa Egitura (LDE diagrama). 9
2.2 Lanaren Deskonposaketa Egitura (LDE diagrama). 11

3.1 Nodoen arteko *topic*-en bidezko komunikazioa. 26
3.2 Bi koordenatu-sistemen arteko erlazioa. 30
3.3 Bi koordenatu-sistemen arteko erlazioa tf bidez definituta. . . 30
3.4 URDF adibide eredua. 31
3.5 Link elementuaren adibide grafikoa. 32
3.6 *Joint* elementuaren adibide grafikoa. 34
3.7 Mundu hutsa Gazebo simulagailuan. 36

4.1 Torresol zentralaren kolektore-segida. 39
4.2 Zentralaren simulazioa Gazebon. 41
4.3 Ingurune simulatuaren exekuzio garaiko nodoak. 41

5.1 Mainbot robot mugikorra. 44
5.2 LMS221 laser sentsorea. 46
5.3 Laser sentsorearen zenbait aplikazio praktiko. 46
5.4 Laser sentsorearen eragiketa-printzipioa. 47
5.5 LMS221 laserraren transmisioaren norabidea. 48
5.6 Mainbot robotaren URDF eredua. 50
5.7 Laser sentsorearen irakurketak RViz-en. 52
5.8 Mainbot robota Gazebo simulagailuan. 53
5.9 Kontroladoreen metodoen exekuzio fluxua. 55
5.10 Kontroladoreen egoera posibleak. 57
5.11 Mekanismo ereduaren robotaren eta simulazioaren konfigurazioa. 58
5.12 Ackermann geometriaren adierazpide grafikoa. 60
5.13 PID kontrol baten bloke-diagrama. 63
5.14 Kontroladorearen garapenaren lehen fasea. 64
5.15 Kontroladorearen garapenaren bigarren fasea. 66

5.16 Kontroladorearen garapenaren azken fasea. 67
5.17 Ingurune simulatuaren eta robotaren simulazioaren exekuzio garaiko nodoak. 69

6.1 Torresol zentralaren nabigaziorako mapa. 76
6.2 *Navigation* pilaren egitura orokorra. 78
6.3 **move_base** nodoaren berreskuratze-portaerak. 79
6.4 Kostu-mapen gelaxken balioen aukeraketa. 82
6.5 Planifikatzaile lokalaren ibilbide posibleak. 85
6.6 Nabigazio-sistemaren bistaratzea RViz tresnan. 89
6.7 Sistema robotikoaren exekuzio proba Gazebon ikusita. 90
6.8 Sistema robotikoaren exekuzio proba RViz-en. 91
6.9 Sistema osoaren exekuzio garaiko nodoak. 93

Taulen Zerrenda

2.1 Proiektuaren denbora plangintza. 10
2.2 Proiektuaren estimatutako denbora eta denbora errealaren arteko konparaketa. 13

5.1 Mainbot robotaren ezaugarri teknikoak. 45
5.2 Laser sentsorearen ezaugarri teknikoak. 48

Laburpena

Proiektu honetan IK4-Tekniker fundazioaren Mainbot robotaren simulazioa eta nabigazioa garatu dira. Mainbot robota zentral termosolar baten mantentze-lanetan lagungarria izateko dago pentsatua, eta Ackermann mugimendu-sistema du. Proiektuaren barnean ondorengoak garatu dira: simulazioaren ingurunea, robot simulatua eta nabigazio-sistema. Robota A puntutik B puntura modu autonomoan nabigatzeko gai izan beharko da, eta, horretarako, sentsoreetatik jasotako informazioa erabili beharko du oztopoak ekiditeko.

1 Kapitulua

Sarrera

Gaien Aurkibidea

1.1 Sarrera 2
1.2 Proiektuaren deskribapen orokorra 2

1.1 Sarrera

Robot bat zer den definitzerako orduan aukera desberdinak daude, ez dagoelako guztiz argi. Hurrengo esaldia har genezake definizio orokor bezala: robot bat mundu fisikoan existitzen den sistema autonomo bat da, eta honek ingurunearen informazioa jaso dezake helburu batzuk lortzeko asmoz. Robot bat autonomoa dela esateak ondorengoa esan nahi du: robota erabakiak hartzeko gai dela eta erabaki horien arabera joka dezakeela, gizakien laguntzarik gabe.

Proiektu honetan Mainbot izeneko robota simulatu da, eta modu autonomoan nabigatzeko gaitasuna eman zaio. Horretarako, ROS robotak maneiatzeko liburutegi-sorta eta Gazebo simulagailua erabili dira, 3. atalean azalduko direnak. Mainbot lurreko robotak Ackermann mugimendu-sistema du eta eremu zabaletan nabigatzeko aproposa da. Bere helburua, instalazio industrialen mantentze-lanetan erabilgarri izatea da, kasu zehatz bakoitzean robotari egin beharko zaizkion aldaketak kontutan hartuta. Proiektu honen garapenean robota zentral termosolar batean kokatu da, etorkizunean beso mugikor bat gehituta bertako ispiluen mantentze-lanetan lagungarri izateko helburuarekin.

Proiektu hau Eibarren kokatutako IK4-Tekniker fundazioan garatu da, fakultatea eta Tekniker erakundearen artean hitzartutako hezkuntza-lankidetza programa bati esker. Fundazio honek ardatz gisa hainbat espezialitate dituen zentro teknologiko bat du, eta, bertan, mekatronika, fabrikazioaren teknologiak eta mikroteknologiak lantzen dituzte. Proiektu hau *Sistemas autónomos e inteligentes* unitatearen barruan egin da.

1.2 Proiektuaren deskribapen orokorra

Esan dugun bezala, proiektu honetan Mainbot robotaren simulazioa eta nabigazioa landu ditugu ROS eta Gazebo erabiliz. Hortaz, bi fase hauetan zati genezake garapena. Helburuak errealitatean existitzen den robot bat simulatzea eta haren nabigazioa garatzea izan dira.

Simulazioari dagokionez, hainbat elementu garatu beharko ditugu. Lehenik eta behin, robotak izango duen ingurunea zehaztuko dugu. Gure kasuan, zentral termosolar bat izango da, eta ROSen URDF lengoaia erabiliko dugu hura zehazteko. Ondoren, robotaren eredu fisikoa definituko dugu, robot mugikorra dela kontutan izanda eta URDF lengoaia erabiliz. Horretarako, higiezinak diren robotaren zatiak identifikatuko ditugu (oinarri mugikorra, gurpilen ardatzak eta gurpilak, nagusiki) eta hauen arteko erlazioak ezarriko ditugu. Azkenik, robotak Ackermann mugimendu-sistema duela kontutan

izanik, robotaren kontroladorea idatzi beharko dugu. Kontroladore honek robotari orokorrean aplikatu behar zaizkion abiadura eta noranzkoa jasota gurpil bakoitzaren abiadura eta norabidea kalkulatu beharko ditu. Robotaren nabigazio-sistema robotaren kokalekutik helburu-puntu batera modu autonomoan eramateko gai izan behar da. Beste hitz batzutan esanda, A puntutik B puntura joateko ahalmena izan behar du. Horretarako, nabigazio-sistemak ingurunea definitzen duen mapa estatiko bat erabiliko du, eta mapa horren arabera planifikatuko du jarraitu beharreko ibilbidea. Hala ere, baliteke oztopo dinamikoak agertzea ingurune horretan (ingurunearen zati ez direnak, alegia) eta hau kontutan izan beharko du nabigazio-sistemak. Helburu-puntua erabiltzaile batek edo sistema informatiko batek ezarri ahalko du.

Memoria honek hainbat atal ditu. 2. atalean proiektuaren helburuen dokumentua ikusiko dugu, proiektuaren kudeaketa azaltzen duena. 3. atalean ROS euskarriaren (*framework* ingelesez) eta Gazebo simulagailuaren deskribapen zehatza ikusiko dugu. Ondoren, ingurune simulatuaren deskribapen labur bat ikusiko dugu 4. atalean. Jarraitzeko, robota errealaren deskribapena eta robot errealaren simulazioa ikusiko ditugu 5. atalean. Proiektuan garatutakoarekin amaitzeko, 6. atalean nabigazio-sistemaren muina ikusiko dugu. Amaitzeko, proiektu hau garatu ondoren ateratako ondorioak eta etorkizunean hau hobetzeko egin daitezkeen lanak zerrendatuko ditugu 7. atalean.

2 Kapitulua

Proiektuaren helburuen dokumentua

Gaien Aurkibidea

2.1 Proiektuaren deskribapena eta helburuak 6
2.2 Proiektuaren abiapuntua 7
2.3 Ataza eta azpiatazen zerrenda 7
2.4 Lanaren Deskonposaketa Egitura (LDE diagrama) 8
2.5 Proiektuaren plangintza 8
2.5.1 Estimazioa 10
2.6 Lan metodologia 11
2.7 Arriskuen analisia 12
2.7.1 Arriskuen zerrenda 12
2.7.2 Arriskuen kontingentzia plana 12
2.8 Denbora erreala 13

Proiektuaren helburuen dokumentuan garatu den softwarearen azalpena egingo da. Proiektuaren deskribapena eta helburuak azaltzeaz gain, plangintza estimazioa, ataza eta azpiatazen zehaztapena, arriskuen analisi eta kontingentzia plana eta lan metodologia azalduko dira. Azkenik, estimatutako denbora eta errealitatean emandako denborak alderatuko ditugu.

2.1 Proiektuaren deskribapena eta helburuak

Proiektu honetan gune zabaletako mantenimendurako robot bat simulatu da ROS/Gazebo softwarea erabiliz. Mainbot robota Robosoft enpresak eraikia da eta RobuCarTT plataforman oinarritzen da.

Proiektu honen helburu nagusia Mainbot robota programatzea eta simulatzea izango da, eta simulazioari esker etorkizuneko programazioa erraztea espero da. Robot hau gune zabaletako instalazioen mantenimenduan modu moldagarri eta malguan erabilgarria izango da. Gure kasuan, energia berriztagarrien instalazio batean kokatuko da, eta instalazio horretan zehar modu autonomoan nabigatzeko gai izan beharko da robota. Horretarako, robota ROS *framework*-ean programatzea erabaki da eskaintzen duen funtzionalitate altua eta baliabide anitzengatik.

Robota simulatzeak beste hainbat abantaila eskaintzen ditu, hala nola: robot errealaren hasieratze denbora luzeak sahiestea (aldi askotan jarraian hasieratu behar denean gehienbat), robotaren operazio kostuak sahiestea matxurei eta halakoei dagokienez eta, kasu honetan bereziki garrantzitsua dena, robotaren helburu ingurunea simulatzea. Azken hau bereziki garrantzitsua da hainbat arrazoi egon daitezkeelako ingurune errealean probak garesti izateko. Adibidez, gure kasuan helburu ingurunea Cadizen kokatutako energia berriztagarrien instalazioa da, eta, nabari den moduan, robotarekin proba bat egin nahi den bakoitzean Cadizera joatea ekonomikoki oso garestia izango litzateke garraiatze eta pertsonal kostuak kontutan izanda.

Robotari dagokionez, Ackermann norabide geometria du eta oztopoen detekziorako eta bere buruaren lokalizaziorako hainbat sentsore ditu. Ackermann norabide geometria izateak “arazo” bat suposatuko du gure kasuan: ROSek eskaintzen duen nabigaziorako softwareak ez ditu murrizketa ez-holonomoak kontuan hartzen modu lehenetsian. Beraz, moldaketa batzuk egin beharko ditugu gure helburuak asebetetzeko. Horretaz aparte, ROS erabiltzeak lana erreztuko digu bera baita robotikako ikerkuntzan ikerlari gehienek erabiltzen duten *framework*-a, eta, ondorioz, baliabide gehien eta atzitzeko errazen dituena.

Robota ROS/Gazebon simulatzeko hainbat osagai garatu beharko ditugu elkarrekiko independenteak izan arren exekuzioan bat egingo dutenak.

Exekuzio unean komunikatzeko ROSek eskaintzen dituen *topic*ak erabiliko ditugu, horrela nodoen exekuzio grafo bat lortuz.

Garatu beharreko osagaiak, hurrenez hurren, ondorengoak dira: instalazio berriztagarrien simulazioa, robotaren simulazioa (kontroladorea barne) eta nabigazio-sistema. Aipatutako osagai hauetatik konplexuenak izango diren zatiak robotaren kontroladorea idaztea eta nabigazio-sistema Ackermann norabide-geometriara egokitzea izango dira.

2.2 Proiektuaren abiapuntua

Proiektu hau egiteko ideia Informatika Ingeniaritzako Sistemen Kontrola ikasgaia egin ondoren sortu zitzaion ikasleari. Robotikaren munduan interesa sortu zitzaiola-eta KBPa enpresa batean egitea erabaki zuen. Horretarako, IK4-Tekniker enpresak eskainitako ikasketa-hitzarmen bati esker 2012ko urritik 2013ko otsailera arte proiektua garatzen jardun zuen. Enpresa honen ezagutza eta esperientzia oso baliagarria izan da garapenaren orduan, bileretan eta bileretatik kanpo pertsonal kualifikatua laguntzeko prest bait zegoen beti.

2.3 Ataza eta azpiatazen zerrenda

Aurkezten den proiektuan bost fase nagusi bereizten dira. Fase hauek softwarea garatzeko kontuan hartu diren atazak biltzen dituzte eta proiektuaren bizi zikloa islatzen dute.

- Proiektuaren kudeaketa.
  - Lana kudeatzeko eta planifikatzeko azpiatazen multzoa.
    - Bileren antolakuntza eta gauzatzea, proiektuaren kontrola, planifikazioa eta kudeaketa, proiektuaren formalizaziorako memoria eta aurkezpen publikorako materiala.
- Analisia.
  - Proiektua garatzeko beharrezko informazioaren bilketa edo zerikusia duten aplikazioen analisia.
  - Garatutako softwarearen arkitektura.
  - Helburuen zehaztapena.
    - Proiektuaren ezaugarriak finkatzea.

  - Arriskuen analisia.

- Inplementazioa.
  - Analizatutako sistemaren inplementazioa burutzea.
    * Softwarea garatzeko teknologien aukeraketa.
    * Ingurunearen simulazioa garapena.
    * Robotaren simulazioa garapena.
    * Nabigazio sistema garapena.

- Probak eta ebaluaketak.
  - Garatu den sistema egiaztatu, zuzendu, baloratu eta hobekuntzak identifikatzea.

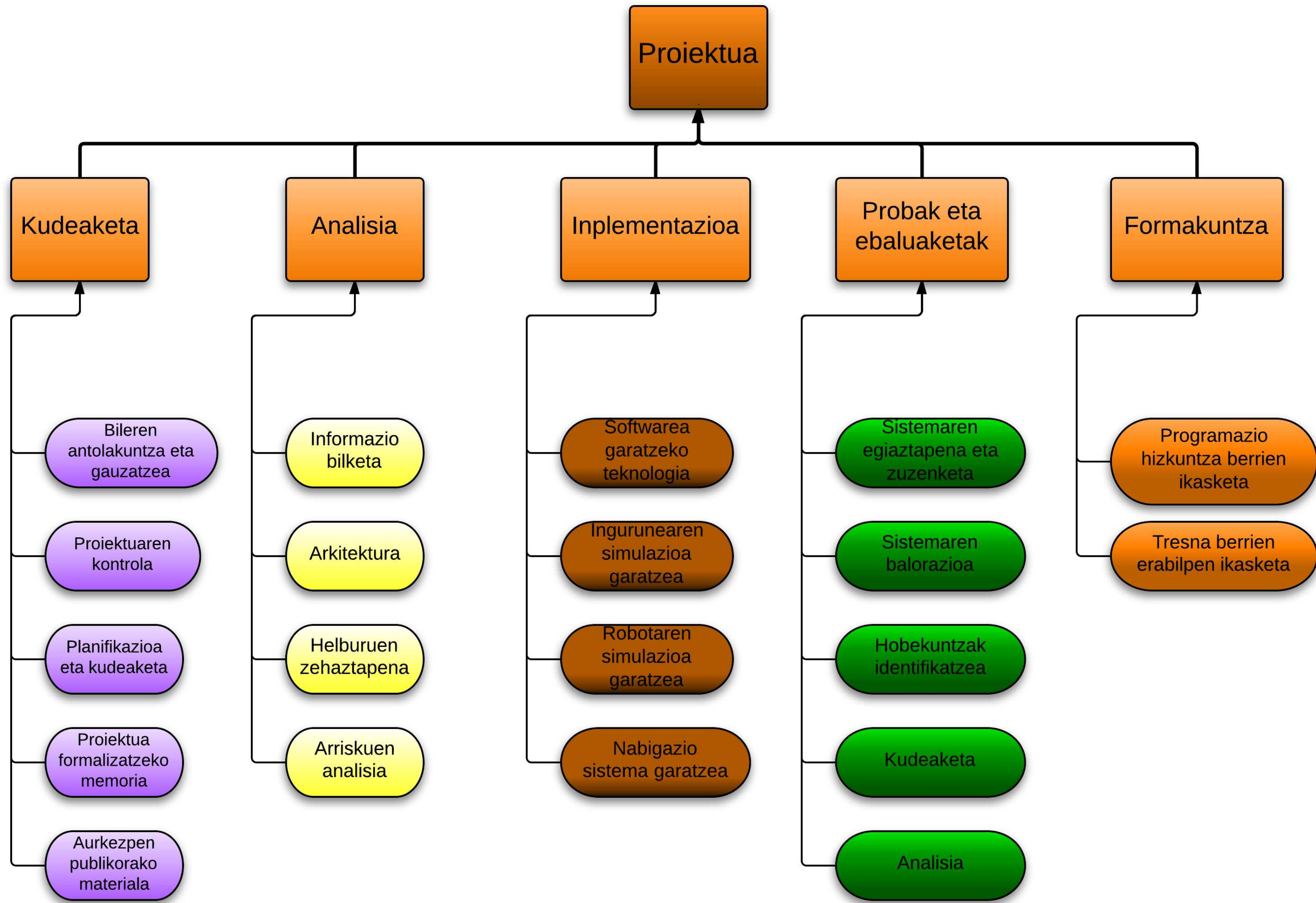
- Formakuntza.
  - Proiektua garatu ahal izateko beharrezko gaitasunak identifikatu eta lortzea.
    * Teknologia berriak barneratzea eta baliagarriak izan daitezkeen tresnen erabilera ikastea.

2.4 Lanaren Deskonposaketa Egitura (LDE diagrama)

Lanaren Deskonposaketa Egitura diagraman proiektuaren norainokoak finkatzen dira era multzokatu, hierarkiko eta egituratuan. Honela, burutu beharreko lan karga atazetan banatzen da era grafikoan. Ataza eta azpiataza hauek aurreko atalean azaldutakoak dira. 2.2 irudian ikusi dezakegu proiektuko LDE diagrama.

2.5 Proiektuaren plangintza

Atal honetan aurkezten den proiektuaren garapena azalduko da denboran zehar, proiektuaren parte diren ataza eta azpiataza guztiak nola garatuko diren finkatuz.



Irudia 2.1: Lanaren Deskonposaketa Egitura (LDE diagrama).

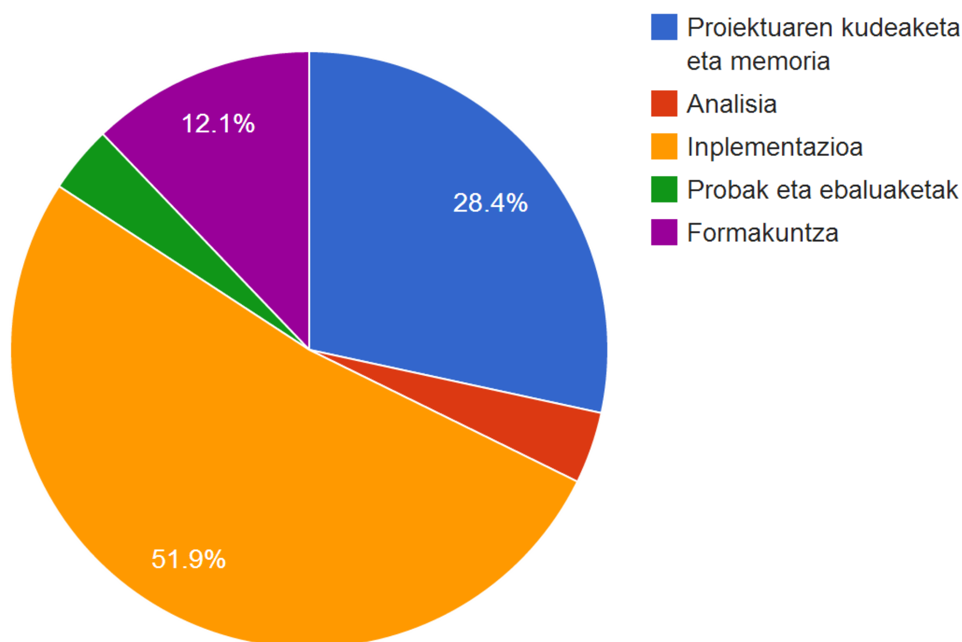
2.5.1 Estimazioa

2.1 taulan ataza eta azpiataza guztientzat egindako denbora estimazioak aurkezten dira. Aurrerago estimatutako denboren desbideraketak aztertu ahalko dira.

Atazak eta azpiatazak	Estimazioak
Proiektuaren kudeaketa eta memoria	**117**
Bileren antolakuntza eta gauzatzea	20
Proiektuaren kontrola	4
Planifikazioa eta kudeaketa	8
Proiektuaren formalizaziorako memoria	75
Aurkezpen publikorako materiala eta prestaketa	10
Analisia	**16**
Informazioaren bilketa eta aplikazioen analisia	7
Garatutako softwarearen arkitektura	5
Helburuen zehaztapena	3
Arriskuen analisia	1
Inplementazioa	**214**
Softwarea garatzeko teknologien aukeraketa	3
Ingurunearen simulazioa garatzea	30
Robotaren simulazioa garatzea	130
Nabigazio sistema garatzea	51
Probak eta ebaluaketak	**15**
Formakuntza	**50**
Ataza eta azpiataza guztien batura	**412**

Taula 2.1: Proiektuaren denbora plangintza.

2.2 irudiko grafikoan ikusi dezakegu grafikoki proiektuaren denbora guztizkoa nola banatzen den ataza desberdinen artean. Lasai esan dezakegu inplementazioa dela proiektuaren denboran pisu handiena izango duen ataza. Ondoren, proiektuaren kudeaketa eta memoria egongo da, aurrekoarengandik nahiko urruti. Formakuntzari dagokionez, aurreikusita dago denbora zati handi bat beharko dela proiektua aurrera eramateko. Azkenik, probak eta ebaluaketa eta analisia atazak denbora gutxi izango dute proiektuaren denbora-banaketan, nahiz eta ataza garrantzitsuak izan.



Irudia 2.2: Lanaren Deskonposaketa Egitura (LDE diagrama).

2.6 Lan metodologia

Proiektua hasi baino lehen, enpresaren behar eta nahiak era orokorrean finkatuko dira lanaren gidalerro moduan.

Lanaren aurrerapenak ikusteko eta kontrolatzeko bilerak egitea erabaki da. Bilera hauek ez dira periodikoki egingo, enpresako lan karga dela eta zalantzarik dagoenerako edo berrikuspenen bat beharrezkoa denerako utziko dira. Hala ere, bilerarik egin gabe denbora luzea igarotea ekidingo da. Hauen iraupena ordu bete ingurukoa izango dela estimatzen da.

Ikasleak enpresako postu bat izango du erabilgarri, eta langile baten 8 ordu eta 21 minutoko jardunaldia izango du egunero. Hala ere, baliteke ez egotea jardunaldi guztia proiektua jorratzen, enpresako beste behar edo ekitaldi batzuetan murgildu daitekeelako.

Lana egiteko enpresak postuaz gain ordenagailu pertsonal bat utziko dio ikasleari, eta honek beharren arabera erabiliko du.

Azkenik, fitxategien kudeaketari dagokionez, babes kopiak automatikoki egiten dituen softwarea erabiliko da.

2.7 Arriskuen analisia

Proiektuaren egikaritzapenean zehar hainbat arrisku sor daitezke. Arrisku hauek softwarea garatzearen ondorio dira eta arrisku hauen aurreikuspena burutzea oso garrantzitsua da. Arriskuen analisiarekin hauek sor ditzaketen kalteak ekidin edota murriztea da helburua, eta horretarako beharrezkoa izango da arriskuak identifikatzea eta dagozkien kontingentzia planak zehaztea.

2.7.1 Arriskuen zerrenda

Proiektuaren inguruan aurreikusten diren arriskuen zerrenda ondorengoa da:

1. Garapeneko fitxategiak galdu daitezke ordenagailuarekin arazorik izanez gero.

2. Bilerak nahi bezain besteko frekuentziarekin egitea zaila gerta daiteke enpresako langileen lan-karga altua dela eta.

3. Ikasleak ezagutzen ez duen software batekin lan egingo duenez honen erabilerarekin trabatuta gera daiteke inplementazioko uneren batean.

2.7.2 Arriskuen kontingentzia plana

Aurreko arriskuen kaltea sahiestu edota murrizteko asmoz ondoko kontingentzia plana diseinatu da:

1. Arrisku hau sahiesteko garatu beharreko softwarearen fitxategien babes-kopiak egingo dira automatikoki. Horretarako Dropbox[1] softwarea erabiliko da. Honek, fitxategi-sisteman zehaztutako direktorio baten azpiko fitxategi eta direktorioak automatikoki gordeko ditu zerbitzari zentral batean, horrela kopia lokalen galera kasuan fitxategiak zerbitzari zentraletik berreskuratu daitezkeelarik.

2. Momentu zehatz batean bilerak egitea zail egiten bada, ikasleak hasierako gidalerroei jarraiki erabakiak hartu beharko ditu, beti ere hartutako erabakien berri eman beharko duelarik aurrerago.

3. Garapen fasean ikaslea tresna baten funtzionalitatearekin trabatzen bada unitateko beste langileei argibideak eskatuko dizkie.

[1] Ikus http://www.dropbox.com webgunea.

2.8 Denbora erreala

Azkenik, estimatutako eta errealitatean erabilitako orduen alderaketa egingo dugu. 2.2 taulan ikus dezakegun bezala, estimatutako ordu kopurua gutxagatik motz geratu da. Ez da desbideraketa nabarmenik eman baina oro har estimatutakoa baina denbora gehiago behar izan da analisia ez ezik beste ataza guztietan.

Atazak eta azpiatazak	Estimazioak	Erreala
Proiektuaren kudeaketa eta memoria	**117**	**127**
Bileren antolakuntza eta gauzatzea	20	15
Proiektuaren kontrola	4	5
Planifikazioa eta kudeaketa	8	7
Proiektuaren formalizaziorako memoria	75	90
Aurkezpen publikorako materiala eta prestaketa	10	10
Analisia	**16**	**15**
Informazioaren bilketa eta aplikazioen analisia	7	8
Garatutako softwarearen arkitektura	5	4
Helburuen zehaztapena	3	2
Arriskuen analisia	1	1
Inplementazioa	**214**	**224**
Softwarea garatzeko teknologien aukeraketa	3	4
Ingurunearen simulazioa garatzea	30	35
Robotaren simulazioa garatzea	130	140
Nabigazio sistema garatzea	51	45
Probak eta ebaluaketak	**15**	**20**
Formakuntza	**50**	**55**
Ataza eta azpiataza guztien batura	**412**	**441**

Taula 2.2: Proiektuaren estimatutako denbora eta denbora errealaren arteko konparaketa.

3 Kapitulua

ROS

Gaien Aurkibidea

3.1 Sarrera **17**
3.1.1 Ezaugarriak 17
3.2 Kontzeptu orokorrak **20**
3.3 Fitxategi-sistema **20**
3.3.1 Paketeak 21
3.3.2 Manifestuak 21
3.3.3 Pilak (*Stack*) 22
3.3.4 Pilen manifestuak (*Stack Manifest*) 22
3.3.5 Mezu-mota (msg) 23
3.3.6 Zerbitzu mota (srv) 23
3.3.7 Launch fitxategiak 23
3.4 Konputazio maila **24**
3.4.1 Nodoak 24
3.4.2 ROS gainbegiralea 24
3.4.3 Parametroen zerbitzaria 25
3.4.4 Mezuak 27
3.4.5 Topicak 27
3.4.6 Zerbitzuak 28
3.5 Goi mailako kontzeptuak **28**
3.5.1 Erreferentzia-sistemak/Transformatuak 29
3.5.2 Robot eredua (URDF) 31

3.6 Simulazioa 33

3.6.1 Gazebo 34

3.1 Sarrera

Robotentzako softwarea idaztea zaila da, batik bat robotikaren eskala eta helburuak hazten ari diren heinean. Robot ezberdinek hardware oso desberdina eduki dezakete, eta honek kodea berrerabiltzea zailtzen du. Gainera, kode asko behar da, maila askotako softwarea behar duelako; kontrolagailu mailako softwaretik hasita pertzepzioa, arrazoibide abstraktua eta gehiago.

Erronka hauei aurre egiteko, robotikako ikerlariek hainbat *framework* sortu dituzte denboran zehar konplexutasuna kudeatzeko eta esperimentuetarako softwarea azkar prototipatzeko. Hauen emaitza dira gaur egun irakaskuntzan eta industrian erabiltzen diren robotikako hainbat software.

ROS bere diseinu zikloan egindako konpromiso eta lehentasunen emaitza da. Bere helburua robotikako ikerkuntzan ahalik eta egoera gehienetan erabilgarria izatea da.

Software honek sistema eragile baten moduko zerbitzuak eskaintzen ditu: hardwarearekiko abstrakzioa, behe-mailako gailuen kontrola, prozesuen arteko mezu-trukaketa eta paketeen kudeaketa. Grafo arkitekturan dago oinarrituta, eta grafoko nodo bakoitza zeregin zehatz baterako prozesaketa egiteko egongo da diseinatua. Unix motako sistema eragileetarako dago garatua, eta, orokorrean, Ubuntu Linux banaketa erabiltzen da.

Lehen bertsioa 2007an argitaratu zen, eta eskuartean duzun proiektua ROS Fuerte bertsioa erabiliz garatu da.

3.1.1 Ezaugarriak

Diseinatutako arkitektura ez dago soilik zerbitzuko roboten eta manipulazio mobileko domeinuetarako pentsatua; ahalik eta orokorrena da. Hauek dira bere diseinuan kontuan edukitako ideia nagusiak:

Puntutik punturakoa

ROS bidez eraikitako sistema bat hainbat prozesuk osatzen dute. Prozesuak makina desberdinetan egon daitezke eta puntutik punturako topologia bidez konektatzen dira exekuzio denboran. Nahiz eta zerbitzari zentral batean oinarritzen diren *framework*-ek (adibidez, CARMEN) hainbat abantaila izan ditzaketen diseinu multiprozesu eta ostalari anitzaren aurrean, zerbitzari zentral batek hainbat arazo eman ditzake konputagailuak sare heterogeneo batera konektatuta badaude.

Adibidez, ROS diseinatzerako orduan kontuan edukitako zerbitzurako robot handietan hainbat konputagailu izaten dira Ethernet bidez konektatuta.

Sare-segmentu hau haririk gabeko LAN bitartez kanpoko makinetara konektatuta dago. Makina hauetan konputazio intentsiboko atazak exekutatzen dira, hala nola konputagailu bidezko ikusmena eta berbaren atzematea. Zerbitzari zentrala sistema barruan edo kanpoan izateak trafiko fluxu handi eta alferrikakoa sortuko luke, hainbat mezu azpisare batean soilik mugitzen direlako. Aldiz, puntutik punturako topologia erabiliz, behar denean software bidezko bufferingarekin konbinatuta, arazo hau sahiesten da.

Puntutik punturako topologiak prozesuen izen-zerbitzari baten beharra du exekuzio denboran prozesuak elkar aurkitzeko gai izan daitezen. Zerbitzu hau **master** bezala ezagutzen da ROSen.

Lengoaia anitzekoa

Kodea idazterako orduan pertsona bakoitzak bere lehentasunak izaten ditu programazio lengoaiei dagokienez. Lehentasun hauek esperientzia pertsonalak, arazketarako erreztasunak, sintaxiak, exekuzio denborako eraginkortasunak... bezalako ezaugarriek baldintzatzen dituzte. Horregatik, lengoaiekiko independentzia du ROS-ek eta, gaur egun, C++, Python, Octave, LISP, Java eta lengoaia gehiagorekin lan egitea posible da.

ROSen espezifikazioa mezuen geruzan geratzen da, ez urrunago. Puntutik punturako konexioaren negoziazioa eta konfigurazioa XML-RPC bidez egiten da (lengoaia gehienetan inplementatuta dago azken hau). C lengoaian oinarritutako aplikazio bat eta lengoaia bakoitzarentzako kode auxiliarra eman beharrean, ROS lengoaia bakoitzean hutsetik inplementatzen da eta horrela lengoaia horren konbentzioak errazago jarraitzen dira.

ROSek, moduluen artean bidalitako mezuak deskribatzeko lengoaiekiko independentea den interfazeen definizio-lengoaia (IDL) erabiltzen du; horrela, hainbat lengoaiatan garatu ahalko dugu softwarea. IDLak testu zati txikiak erabiltzen ditu mezu bakoitzaren eremuak zehazteko, eta mezuen arteko konposizioa ahalbidetzen du. Adibidez:

```
Header header
Point32[] pts
ChannelFloat32[] chan
```

Babestutako lengoaia bakoitzeko konpilatzaileek bertako objektuak bezala izango diren inplementazioak sortuko dituzte. ROSek automatikoki serializatuko eta deserielizatuko ditu mezu bat bidaltzen edo jasotzen denean. Horrela, programatzaileak denbora galtzea eta erroreak egitea saihesten da: aurreko 3 kode-lerroak C++en 137 lerrotara zabaltzen dira, Pythonen 96era, eta, Lispen, 81ra.

Emaitza bezala lengoaiarekiko independentea izango den mezu prozesu eskema bat izango dugu, eta, bertan, hainbat lengoaia batu eta konbinatu ahalko ditugu.

Tresnetan oinarritua

ROSek microkernel diseinua jarraitzen du bere konplexutasuna murrizteko. Garapenerako eta exekuziorako ingurune monolitiko bat sortu beharrean, hainbat tresna txiki erabiltzen dira ROSeko osagaiak sortu eta exekutatzeko.

Tresna hauek hainbat ataza betetzen dituzte: iturburu-kodearen zuhaitza nabigatu, konfigurazio parametroak jaso eta ezarri, puntutik punturako konexio topologia ikusi, banda-zabaleraren erabilera neurtu, mezuen informazioa grafikoki ikusi, dokumentazioa automatikoki sortu, eta gehiago.

Argala

Robotikan erabiltzen diren software desberdinek, berrerabilgarriak izan beharko luketen kontrolatzaile eta algoritmoak erabiltzen dituzte. Zoritxarrez, kode hauen konplexutasuna dela eta oso zaila izaten da beraien funtzionalitatea ateratzea eta berezko ingurunetik kanpo berrerabiltzea. ROSen konpilazio sistemak zuhaitz egitura jarraitzen duen konpilazio modularra egiten du. Gainera, CMake erabiltzen du ROSek helburu duen egitura argala mantentzeko. Konplexutasuna liburutegietan mantentzen da eta liburutegien funtzionalitatea ROSen ikustarazteko exekutagarri txikiak behar dira; horrela, kodea berezko erabileratik haratago berrerabiltzea asko errazten da.

Gainera, ROSek hainbat kode irekiko proiektuetako kodea berrerabiltzen du: kontroladoreak, nabigazio-sistemak, Player proiektuko simuladoreak, OpenCV-ko ikusmen algoritmoak, OpenRAVEko planifikazio algoritmoak, eta gehiago. Kasu bakoitzean, konfigurazio aukerak ikusteko edo datuak software batetik barrura edo kanpora zuzentzeko erabiltzen da ROS, ahalik eta *wrapping* gutxiena erabiliz.

Askea eta kode irekia

ROSen iturburu-kode guztia publikoa da. Horrela, softwarearen arazketa errazten da maila guztietan; bereziki hardwarea eta softwarea paraleloan arazten direnean. ROS BSD lizentziarekin banatzen da, eta horrek proiektu komertzial naiz ez komertzialak garatzeko erabiltzea ahalbidetzen du. Modulen arteko datu-trukaketak prozesuen arteko komunikazioak erabiliz egiten dira, eta, horregatik, ez da beharrezkoa moduluak exekutagarri berdinean integratuta egotea. Gauzak horrela eta filosofia modularrari jarraiki, pakete

bat osatzen duten osagai ezberdinek lizentzia desberdinak izan ditzakete, GPLtik hasita BSDra arte.

3.2 Kontzeptu orokorrak

ROSen oinarrizko kontzeptuak **nodoak**, **mezuak**, **topicak** eta **zerbitzuak** dira.

Nodoak konputazioa egiten duten prozesuak dira. ROS eskala minimoan modularra izateko diseinatua dago: sistema bat orokorrean hainbat nodok osatzen dute. Testuinguru honetan, **nodo** terminoa **software modulu**gatik aldatu daiteke. Nodo terminoa erabiltzeak ROSen oinarrituta dauden sistemen exekuzio denborako bistaratzea laguntzen du: hainbat nodo exekuzioan daudenean, puntutik punturako komunikazioa grafo bat bezala ikusi behar da, prozesuak nodoak izango direlarik eta, loturak, grafoko arkuak.

Nodoak elkarren artean mezuak pasatuz komunikatzen dira. Mezu bat gogorki tipatutako datu-egitura da. Oinarrizko datu-egiturak (osokoak, errealak, boolearrak, etc.) erabilgarri daude, eta datu-mota hauen eta konstanteen taulak ere bai. Mezuak elkarren artean konposatu daitezke mezu berriak sortzeko, eta mezuen taulekin ere egin daiteke konposaketa, nahi den mailan.

Nodo batek izendun *topic* batean mezuak publikatuko ditu. Datu konkretu batzuetan interesatuta dagoen nodo bat *topic* egokira harpidetuta egongo da. *Topic* bakar batean une jakin batean hainbat argitaratzaile edo harpidedun egon daitezke konkurrenteki, eta nodo bakar bat nahi adina *topic*-etara argitaratu edo harpidetu daiteke.

Topic-etan oinarritutako argitaratzaile-harpidedun eredua komunikazio paradigma malgua den arren, ez da egokia transakzio sinkronoetan erabiltzeko. ROSen zerbitzu izena ematen zaio honi, eta karaktere-kate batez eta bi mezu-moten bitartez identifikatzen dira: bat eskaeretarako eta bestea erantzunetarako. *Topic*-ekin ez bezala, nodo bakar batek argitaratu dezake zerbitzu jakin bat.

3.3 Fitxategi-sistema

ROSen fitxategi-sistema diskoan aurkitzen diren baliabideek osatzen dute. Adibidez: **paketeak**, **manifestuak**, **pilak**, **stack manifestuak**, **mezu** (msg) motak, **zerbitzu** (srv) motak... Hauek denak deskribatuko ditugu jarraian.

3.3.1 Paketeak

ROSen softwarea paketetan multzokatzen da. Pakete batek hainbat elementu izan ditzake: nodoak, liburutegiak, datu-multzoak, konfigurazio-fitxategiak, ROSena ez den software zati bat, edo modulu erabilgarri bat izango den beste edozer gauza. Pakete hauen helburua kodea berrerabiltzea posible izango den modu batean funtzionalitatea ematea da. Orokorrean, ROS paketeek *Goldilocks* printzipioa jarraitzen dute: erabilgarria izateko adina funtzionalitate, baina ez gehiegi, berrerabiltzeko zaila izan ez dadin.

Paketeak sortzea oso erraza da; eskuz edo *roscreate-pkg* izeneko tresna baten bidez sortu daitezke. ROS pakete bat `ROS_ROOT` edo `ROS_PACKAGE_PATH`en azpian dagoen direktorio bat da. Paketeak elkarren artean pilatan (*stack*) bidez antolatzen dira.

Paketeak ROSen fitxategiak antolatzeko moduan kontzeptu oso garrantzitsuak dira. Hortaz, hainbat tresna daude hauekin lan egiteko, horien artean:

- **rospack:** paketeei buruzko informazioa aurkitu eta eskuratu. Konpilazio sistemak ere *rospack* erabiltzen du paketeak aurkitzeko eta bere dependentziak konpilatzeko.
- **roscreate-pkg:** pakete berri bat sortu.
- **rosmake:** pakete bat eta bere dependentziak konpilatu.
- **rosdep:** pakete baten dependentziak instalatu.
- **rxdeps:** pakete baten dependentziak grafo bezala ikusi.

3.3.2 Manifestuak

Manifestu bat (*manifest.xml*) pakete bati buruzko espezifikazio minimoa da, eta hainbat tresnetan erabiltzen da (konpilazioan, dokumentazioan edota banaketan) ROSen. Pakete baten metadatuei buruz espezifikazio minimo bat emateaz aparte, manifestuen funtzio garrantzitsu bat dependentziak lengoiarekiko eta sistema eragilearekiko neutrala izango den modu batean deklaratzea da. Direktorio batek *manifest.xml* fitxategi bat ezinbestekoa da: fitxategi hau egoteak, ROSen paketeen bide-izenaren azpiko edozein direktoriotan azken hau pakete bezala identifikatzea eragingo du.

Manifest fitxategi batek izan behar duen minimoa oso sinplea da, paketearen autorea eta aplikatutako lizentzia idaztearekin nahikoa da. Lizentzia ezartzea garrantzitsua da, gehienbat ROSen kodea paketeen bidez banatzen delako. Manifestu fitxategi ohikoek `<depend>` eta `<export>` etiketak izaten dituzte, pakete baten instalazioan eta erabileran laguntzeko.

`<depend>` etiketak pakete bat erabiltzeko instalatu beharreko ROS pakete bati egiten dio erreferentzia. Erreferentziatutako paketearen edukiaren arabera esanahi desberdina eduki dezake. Aldiz, `<export>` etiketak lengoiarekiko dependenteak diren konpilazioko eta exekuzioko etiketak deskribatzen ditu, eta pakete horretan oinarritzen den edozein paketek beharko dituenak.

rospack tresnak *manifest.xml* fitxategiak sintaktikoki aztertzen ditu eta hauetatik informazioa ateratzeko ere balio du. Adibidez, `rospack depends pakete-izena` komandoak `pakete-izena` paketearen dependentzia guztiak bistaratuko ditu.

3.3.3 Pilak (*Stack*)

ROSen paketeak pilatan antzolatzen dira. Paketeen helburua berrerabiltzeko errazak izango diren kodearen kolekzio minimoak sortzea den bitartean, pilen helburua kode banaketaren prozesua sinplifikatzea da. Pilak dira softwarea banatzeko gehien erabiltzen den mekanismoa ROSen. Pila bakoitzak bertsio zenbaki bat dauka eta beste pilekiko dependentziak izan ditzake. Dependentzia hauek ere bertsio zenbaki bat izaten dute, garapenerako orduan egonkortasun gehiago izateko.

Pilek kolektiboki funtzionalitate bat ematen duen paketeak elkartzen dituzte, nabigazio paketea edo manipulazio paketea kasu. Software liburutegi tradizionalak konpilazio garaian estekatzen diren arren, pila hauek exekuzio garaian ere eman dezakete funtzionalitatea *topic* edo zerbitzuen bitartez.

Pilak eskuz erraz sortzen dira. Pila bat `stack.xml` fitxategi bat duen eta `ROS_ROOT` edo `ROS_PACKAGE_PATH` azpian dagoen edozein direktorioa da. Direktorio horren barruko edozein pakete pilaren parte dela kontsideratzen da. Gainera, gomendagarria da `CMakeLists.txt` eta `Makefile` fitxategiak pilaren erroan izatea. `roscreate-stack` tresnak automatikoki sortzen ditu fitxategi hauek.

Pilekin lan egiteko tresna garrantzitsuena `rosstack` da. Pakete mailako `rospack` tresnaren analogoa da.

3.3.4 Pilen manifestuak (*Stack Manifest*)

Pilen *manifest* fitxategia (*stack.xml*) ROS pilen espezifikazio minimoa da, eta distribuziorako eta instalaziorako erabiltzen da. Pilaren metadatuak emateaz aparte, *stack.xml* fitxategien eginkizun garrantzitsuenetako bat beste pilekiko dependentziak ezartzea da. Direktorio baten barruan *stack.xml* fitxategi bat izateak ROS direktorio hori pila bezala tratatzea ekarriko du, eta horren barruko edozein pakete pilaren parte bezala tratatuko da.

stack.xml fitxategiaren eduki minimoa *readme* fitxategi bat bezalakoa da, pilaren arduraduna eta pilaren lizentziaren berri ematen du. Lizentzia garrantzitsua da, kodea nork erabili dezakeen esaten duelako. *stack.xml* fitxategiek normalean `<depend>` etiketak izaten dituzte, pila bat instalatu aurretik beste pila batzuk instalatu behar direla jakiteko.

3.3.5 Mezu-mota (msg)

ROSek nodoek publikatzen dituzten datuen balioak deskribatzeko mezuen deskribapen lengoaia sinplifikatu bat erabiltzen du. Deskribapen sinple honekin ROSeko ttresnek hainbat lengoiaia desberdinetarako mezuen kodea modu erraz batean sortzen dute. Mezuen deskribapenak `.msg` motako fitxategietan eta paketeen `msg/` azpidirektorioetan gordetzen dira.

Bi atal ditu `.msg` fitxategi batek: eremuak eta konstanteak. Eremuak mezu barruan bidaltzen diren datuak dira. Konstanteek, eremu horiek interpretatzeko aldagai erabilgarriak definitzen dituzte.

`rosmsg` tresnak mezu-mota baten informazioa inprimatzen du, eta mezu-mota bat erabiltzen duten kode fitxategiak aurkitu ditzake.

3.3.6 Zerbitzu mota (srv)

ROSek zerbitzu motak deskribatzeko zerbitzu-lengoaia sinplifikatua ("srv") erabiltzen du. Hau *msg* formatuan oinarritzen da nodoen arteko eskakizun/erantzun komunikazioa ahalbidetzeko. Zerbitzuen deskribapenak `.srv` fitxategietan gordetzen dira, paketeen `srv/` azpidirektorioetan.

`rossrv` tresnak zerbitzuen deskribapena, `.srv` fitxategiak dituzten paketeak, eta zerbitzu mota bat erabiltzen duten paketeak aurkitzeko balio du.

Zerbitzuak deskribatzeko *msg* motako eskakizun eta erantzun bat definitu beharko ditugu, `"---"` karaktereekin banatuta. Adibidez:

```
string str
 ---
string str
```

3.3.7 Launch fitxategiak

Launch fitxategiak (*.launch*) *roslaunch* tresnak eskaintzen dizkigun baliabideak dira. Fitxategi hauen bitartez hainbat nodo batera exekutatu daitezke, baita parametroen zerbitzarian parametroak ezarri (nodoak eta parametroen zerbitzaria aurrerago azalduko ditugu). Beraz, exekuzioari begira hasieratzea

automatizatzeko tresnak dira. Proiektuan zehar asko erabili den baliabidea da hau, lana asko errazten duelako.

3.4 Konputazio maila

Konputazio-grafoa elkarrekin datuak prozesatzen ari diren prozesuek sortzen duten puntutik punturako sarea da. Grafo hau osatzen duten oinarrizko kontzeptuak nodoak, gainbegiralea (*master*), parametroen zerbitzaria, mezuak, *topic*-ak eta zerbitzuak dira.

3.4.1 Nodoak

Nodo bat konputazioa burutzen duen prozesu bat da. Nodoak grafo egitura batean konbinatzen dira elkarrekin, eta haien artean komunikatzeko ondorengo mekanismoak erabiltzen dituzte: *topic*-ak, RPC zerbitzuak eta parametroen zerbitzaria. Nodoak orokorrean ataza oso konkretuak egiteko daude pentsatuta; robot baten kontrol-sisteman, normalean, hainbat nodo izango dira. Adibidez, nodo batek laserra kontrolatuko du, beste batek robotaren mugimendu-sistema, beste nodo batek lokalizazioa kalkulatuko du, beste nodo batek jarraitu beharreko bidea kalkulatuko du, eta abar.

Nodoak erabiltzeak hainbat onura dakartza orokorrean sistemarentzat. Erroreekiko tolerantzia handiagoa da sistema honen bidez, erroreak nodoetan gertatzen direlako eta, hortaz, nodo batek huts egiteak ez du sistema osoa konpromezuan jarriko beti. Kodearen konplexutasuna ere txikiagoa da sistema monolitikoekin konparatzen badugu. Inplementazioaren xehetasunak ezkutuan geratzen dira nodoek grafoari API minimo bat erakusten diotelako, eta inplementazioak, nahiz eta beste lengoaia batean, modu erraz batean aldatu daitezke.

Exekuzioan dagoen nodo orok unibokoki identifikatuko duen grafoko baliabide izen bat izango du. Adibidez, laser ekorketak bidaltzen dituen Hokuyo kontroladore batek `/hokuyo_node` izena eduki dezake.

Nodo bat ROSeko bezero-liburutegi baten bidez idazten da, hala nola `roscpp` eta `rospy`.

3.4.2 ROS gainbegiralea

ROS gainbegiraleak ROS sistema bateko beste nodoei izen-zerbitzari eta erroldatze zerbitzuak ematen dizkie. Argitaratzaile eta harpidedunak *topic*-etara eta zerbitzuetara gidatzen ditu. Gainbegiralearen eginkizuna nodoak

elkarren artean aurkitzeko gai izatea ahalbidetzea da. Nodoek, elkar aurkitu ondoren, puntutik punturako komunikazioa erabiliko dute.

ROS gainbegiraleak izen-zerbitzari funtzioa du konputazio-grafoan. *Topic*-en eta zerbitzuen erregistro-informazioa gordetzen du nodoentzako. Nodoak gainbegiralearekin komunikatzen dira haien erregistro-informazioa partekatzeko. Nodo hauek gainbegiralearekin komunikatzen direlarik, erregistratutako beste nodoen informazioa jaso dezakete, eta hauekin konexioak egin hala nahi izanez gero. Informazio-erregistroa aldatzen denean gainbegiraleak nodo hauei *callback*-en bitartez jakinaraziko die.

Nodoak elkarren artean zuzenean konektatzen dira, gainbegiraleak kontsultarako informazioa soilik ematen duelarik, DNS zerbitzari baten antzera. *Topic* batera harpidetzen diren nodoek *topic* hori publikatzen duten nodoei konexio-eskaerak egingo dizkiete, eta konexioa gauzatzeko elkarren artean adostutako protokolo bat erabiliko dute. Gehien erabiltzen den protokoloa TCPROS da, eta honek TCP/IP socket estandarrak erabiltzen ditu.

Gainbegiralearen funtzioa hobeto ulertzeko adibide batzuk ikusiko ditugu ondoren. Jo dezagun bi nodo ditugula, bat kamera batetik jasotako irudia publikatzen duena eta, bestea, irudi hori irakurri eta prozesatuko duena. Lehenik, 3.1(a) irudian ikusten den bezala, irudia publikatzen duen nodoak gainbegiraleari `images` *topic*-ean publikatu nahi duela esango dio.

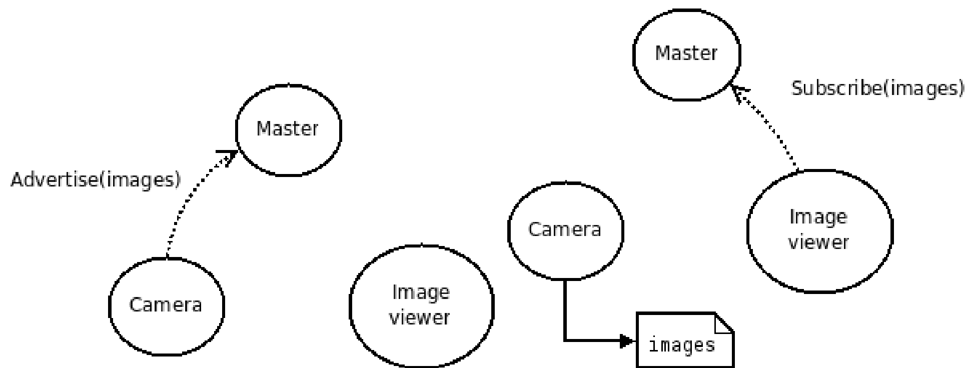
Honela, `camera` nodoak `images` *topic*-ean publikatzen du, baina *topic* horri entzuten dion nodorik ez dagoenez, ez da benetako datu transmisiorik emango. Ondoren, 3.1(b) irudian ikusten den bezala, `Image_viewer` nodoak `images` *topic*-era harpidetu nahi duela esango dio gainbegiraleari, *topic* horretan daturik dagoen jakiteko.

Orain, `images` *topic*-ak, argitaratzaile eta harpideduna, biak, dituenez, gainbegiraleak `Camera` eta `Image_viewer` nodoak bata bestearen existentziaz ohartaraziko ditu, elkarren artean datuak transferitu ditzaten (ikus 3.1(c) irudia).

3.4.3 Parametroen zerbitzaria

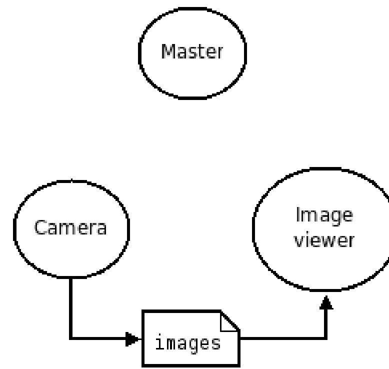
Parametroen zerbitzaria sareko APIen bitartez erabili daitekeen hiztegi partekatu eta multibariatua da. Nodoek zerbitzari hau erabiltzen dute exekuzio-denboran parametroak gordetzeko edo eskuratzeko. Bere helburua eragin-kortasun handiz lan egitea ez denez, konfigurazio-parametroen antzeko datu estatikoak partekatzeko erabiltzea da egokiena. Ikusgarritasun globala du tresnak sistemaren konfigurazio egoera ikusi, eta, beharrezko kasuan, aldatzeko gai izan daitezen.

Parametroen zerbitzaria XMLRPC bidez dago inplementatua eta ROS gainbegiralearen barnean exekutatzen da. Hainbat datu-mota ezberdin era-



(a) *Topic* berri bat sortzeko nodo baten eskaera.

(b) Nodo baten *topic* batera harpidetzeko eskaera.



(c) *Topic* baten bidezko bi nodoen konexioa.

Irudia 3.1: Nodoen arteko *topic*-en bidezko komunikazioa.

biltzen ditu, adibidez:

- 32-bit osokoak
- boolearrak
- karaktere-kateak
- *double*-ak
- iso8601 datak
- listak
- base64 oinarrian kodetutako datu bitarrak

Parametroak ROSen izendatzeko konbentzioak jarraituz izendatzen dira. Horren arabera parametroek *topic* eta nodoetan erabiltzen diren izenespazioen hierarkia jarraitzen dute. Hierarkia hau parametroen izenak beste baliabide baten berdinak izan ez daitezen finkatzen da.

`rosparam` tresna oso lagungarria da parametroekin lan egiteko, adibidez, sortu edo aldatu.

3.4.4 Mezuak

Nodoak elkarren artean komunikatzeko mezuak *topic*-etara argitaratzen dituzte. Mezua datu-mota sinplea da, motadun eremuak dituena. Oinarrizko datu-motak (integer, errealak, boolearrak, etab.) erabiltzeaz gain, hauen zerrendak ere erabili daitezke. Egiturak eta zerrendak nahi bezala konbinatu daitezke mezu baten beharretara egokitzeko.

3.4.5 Topicak

Topic-ak nodoek mezuak trukatzeko erabiltzen dituzten bus izendunak dira. Mezuak modu anonimoan argitaratzen edo entzuten dira, eta, horrela, informazioaren produkzioa eta kontsumoa ezberdintzen dira. Orokorrean, nodo batek ez daki zein beste nodorekin ari den komunikatzen. Horren ordez, informazio zehatz batekiko interesa duten nodoak *topic* egokira harpidetuko dira; informazioa sortzen duten nodoek, aldiz, *topic* egokian argitaratuko dute. *Topic* batek hainbat argitaratzaile edo harpidedun izan ditzake.

Topic-ak streaming bidezko norabide bakarreko komunikaziorako daude pentsatuta. Eskaera/erantzuna antzeko komunikazio bat behar duten nodoek zerbitzuak erabili beharko dituzte.

Topic-ak gogorki tipatuta daude lehen esan dugun mezu-motekin. Horrek esan nahi du *topic* horretan soilik mezu-mota konkretu bat argitaratu daitekeela, eta entzuten dagoen edozein nodok bermatua du ez duela beste motako mezurik jasoko. Gainbegiraleak ez du argitaratzaileen datuen kontsistentzia egiaztatuko. Aldiz, harpidedun nodoek ez dute konexiorik gauzatuko esperotako mezu-mota eta jasotakoa berdinak direla ziurtatu arte.

Mezuen garraiorako TCP/IP eta UDPn oinarritzen da ROS. TCP/IPn oinarritutako garraioa TCPROS bezala ezagutzen da, eta mezuak TCP/IP konexio iraunkorren bitartez bidaltzen ditu. TCPROS da ROSen lehenetsitako garraioa eta liburutegi bezeroek beharrezkotzat duten bakarra. UDPn oinarritutako garraioak, UDPROS bezala ezagutzen dena, mezuak UDP paketeetan banatzen ditu. UDPROS latentzia baxuko garraio galeraduna da, eta, ondorioz, teleoperazioa bezalako atazetarako da egokia.

Nodoek exekuzio garaian negoziatzen dute erabiliko den garraioa. Adibidez, nodo batek UDPROS nahiago badu baina beste nodoak ez badu onartzen, TCPROS garraioa erabili daiteke.

Topic-ekin lan egiteko tresnarik nabarmenena `rostopic` da. Bere erabilera oso erraza da, adibidez:

- `rostopic list` komandoak *topic*-en zerrenda inprimatuko du pantailan.

- `rostopic echo /topic_izena` komandoak `/topic_izena` *topic*-ean transferitutako mezuak inprimatuko ditu pantailan (*topic* horretara harpidetuz).

3.4.6 Zerbitzuak

Argitaratzaile/harpidedun eredua komunikazio paradigma malgua da, baina norabide bakarreko eta hainbat igorle naiz hartzaile izan ditzakeen garraioa ez da egokia eskaera/erantzun motako RPC interakzioetarako (hauek sarri behar izaten dira sistema banatuetan). Eskaera/erantzunak zerbitzuen bidez egiten dira, eta hauek bi mezurekin definitzen dira: bat eskaeretarako eta beste bat erantzunetarako. ROS nodo batek izen bat duen zerbitzu bat eskainiko du, eta bezero batek zerbitzuari deituko dio eskaera-mezua bidaliz eta erantzuna itxaronez. Liburutegi-bezeroek normalean interakzio hau urruneko prozedura baten dei gisa aurkezten dute.

Lehenago esan dugun moduan, zerbitzuak `.srv` fitxategietan definitzen dira.

`rossrv` tresna erabili daiteke `.srv` fitxategietan dauden datu-egiturak ikusteko. `rosservice` tresnarekin abiatutako zerbitzuen zerrenda ikusi dezakegu, edo zerbitzu bati dei bat egin.

3.5 Goi mailako kontzeptuak

ROS plataformaren nukleoa kontrol-arkitekturarekiko ahal den idenpendenteena da. Datuak komunikatzeko hainbat bide eskaintzen ditu (*topic*-ak, zerbitzuak, parametroen zerbitzaria), baina ez du esaten nola erabili behar diren edo nola izendatu behar diren. Ikuspegi honi esker ROS hainbat arkitekturatan integratu daiteke, baina maila altuagoko kontzeptu batzuk beharrezkoak dira ROSen gainean sistema handiak sortzeko. Proiektu honen garapenean oinarrizko diren kontzeptuak soilik azalduko gira jarrian:

3.5.1 Erreferentzia-sistemak/Transformatuak

Sarrera

Sarritan sistema robotikoek erlazio espazialak jarraitu behar izaten dituzte hainbat arrazoigatik: robot mugikor bat eta erreferentzi puntu finko baten arteko lokalizaziorako, sentsoreen koordenatuen eta manipulatzaileen koordenatuen artean, edo kontrolerako objektu helburuetan koordenatuak ezartzeko.

Koordenatu espazialen tratamentua sinplifikatzeko eta bateratzeko *tf* izeneko transformazio-sistema du ROSek. *tf* sistemak sistema bateko koordenatu sistema guztiak erlazionatzen dituen transformatu-zuhaitz dinamiko bat sortzen du. Robotaren azpisistemetatik (kodegailuak, lokalizazio algoritmoak, etab.) informazio jarioa *tf* sistemara sartzen den heinean honek transformatuak argitaratuko ditu.

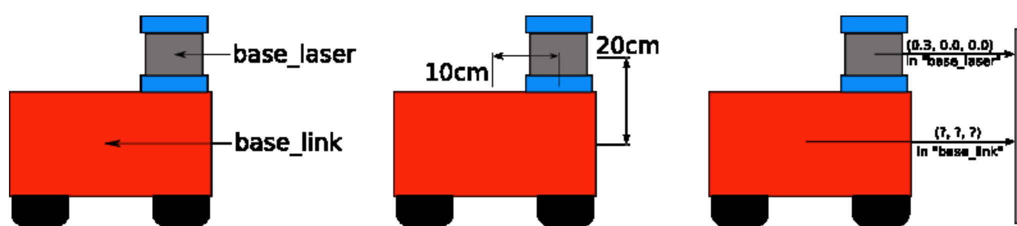
TF paketea

Hainbat ROS paketek robot baten transformatu-zuhaitza *tf* software liburutegiaren bitartez publikatuta egotea behartzen dute. Maila abstraktu batean, transformatu-zuhaitzak erreferentzia-sistema batetik bestera pasatzeko aplikatu beharreko translazio eta errotazio informazioa gordetzen du. Adibide baten bidez ulertzeko, eman dezagun laser bat gainean duen oinarri mugikor sinple bat dugula. Bi erreferentzia-sistema definituko ditugu: bat oinarriaren zentruan dagoen puntua eta, bestea, gaineko laserraren zentruan dagoena. Hauek identifikatzeko izenak emango dizkiegu: *base_link* oinarriaren erreferentzia-sistema izendatzeko (aurrerago ikusiko dugun nabigaziorako garrantzitsua da hau robotaren errotazio-zentruan ezartzea), eta *base_laser*, laserraren zentrua izendatzeko.

Pentsa dezagun laserrak emandako datu batzuk ditugula eta datu hauek laserraren zentruarekiko distantziak direla. Beste modu batera esanda, *base_laser* koordenatu-sisteman datu batzuk ditugula. Orain, datu hauek hartu eta oinarri mugikorrak mundu errealean oztopoak ekiditeko erabili nahi ditugu. Hau ondo egiteko, *base_laser* koordenatu-sistematik jaso ditugun laser ekorketak oinarri mugikorraren *base_link* koordenatu-sistemara pasatu behar ditugu. Finean, 3.2 irudian ikusten den bezala, *base_laser* eta *base_link* sistemen arteko erlazioa definitu behar dugu.

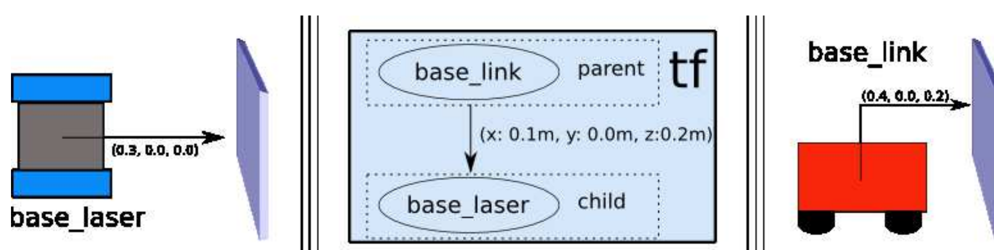
Erlazio hau definitzeko, laserra oinarriaren zentrutik 10 zentimetro aurrerago eta 20 zentimetro gorago dagoela dakigu. Honek, *base_laser* eta *base_link* artean aplikatu beharreko informazio translazionala ematen digu.

Bi koordenatu sistema hauen arteko erlazioa *tf* bidez erabiltzeko eta gordetzeko transformatu-zuhaitz batera gehitu behar ditugu. Kontzeptualki,



Irudia 3.2: Bi koordenatu-sistemen arteko erlazioa.

transformatu-zuhaitzeko nodo bakoitza koordenatu-sistema bati dagokio, eta ertz bakoitzak nodo batetik beste nodo batera joateko aplikatu beharreko transformatua adieraziko du. Edozein bi koordenatu-sistema ertz bakar batez komunikatuta egongo direlako bermatzeko erabiltzen du *tf*-k zuhaitz egitura, eta zuhaitzeko ertz guztiak arbasoetatik umeetara joaten dira.



Irudia 3.3: Bi koordenatu-sistemen arteko erlazioa tf bidez definituta.

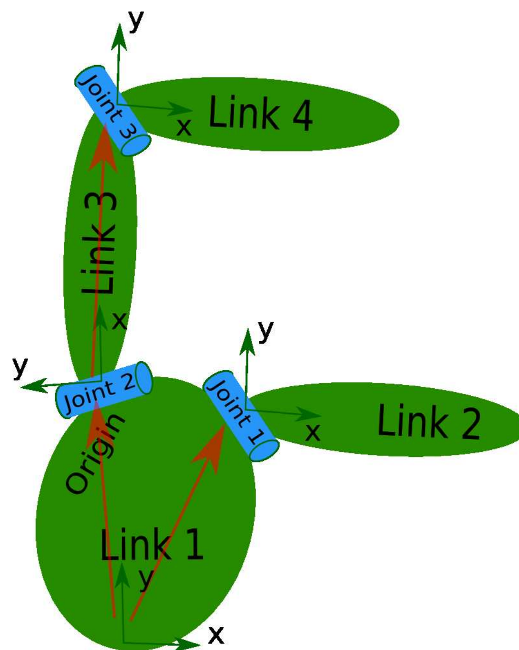
Gure eredu sinplerako transformatu-zuhaitza sortzeko bi nodo sortuko ditugu, bata *base_link* koordenatu-sistemarako eta, bestea, *base_laser* koordenatu-sistemarako. Bien arteko ertza sortzeko, lehenik gurasoa zein izango den eta umea zein izango den erabaki behar dugu. Hau garrantzitsua da ertz guztiak gurasotik umeetara joaten direlako. Guraso bezala *base_link* koordenatu-sistema aukeratuko dugu, pieza edo sentsore gehiago gehitzen diren heinean zentzu gehiago duelako *base_laser* nodoko datuak *base_link* nodoaren bitartez jasotzea. Gauzak horrela, bi nodoen artean aplikatu beharreko transformatua (x: 0.1m, y: 0.0m, z: 0.2m) izango litzateke. Behin transformatu-zuhaitza definituta, laserretik jasotako ekorketak oinarriaren nodora pasatzea *tf* liburutegiari dei bat egitea bezain erraza da. Gure robotak informazio hau erabili dezake oinarria ingurunean oztoporik jotzeko beldurrik gabe ibiltzeko gai izan dadin.

3.5.2 Robot eredua (URDF)

Unified Robot Description Format (URDF) robotak adierazteko XML formatua da. Bere osagai nagusiak *link*-ak eta *joint*-ak dira. *Link*-ei dagokienez, robotaren zati zurrunak adierazten dituzte, eta *joint*-ek, berriz, parte zurrunen arteko loturak. URDF fitxategi baten egitura orokorra ondorengoa da:

```
1 <robot name="test_robot">
2   <link name="link1" />
3   <link name="link2" />
4   <link name="link3" />
5   <link name="link4" />
6
7   <joint name="joint1" /> <!-- link1 eta link2 arteko jointa/>
8   <joint name="joint2" /> <!-- link2 eta arteko jointa/>
9   <joint name="joint3" /> <!-- link3 eta link4 arteko jointa/>
10 </robot>
```

3.4 irudian adibidea grafikoki adierazita ikusi dezakegu.



Irudia 3.4: URDF adibide eredua.

Link elementua

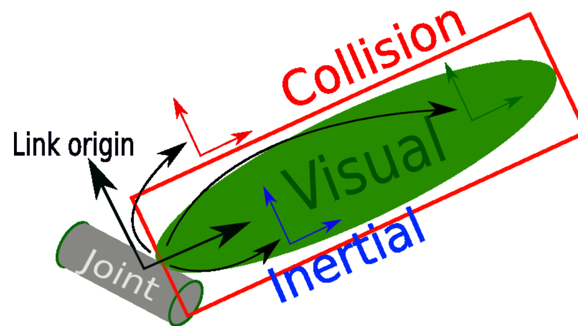
`Link` elementu bat definitzerako orduan, hainbat elementu definitu ditzakegu bere barnean:

- **Inertial:** elementu honen barnean *link*-aren propietate inertzialak definituko ditugu: hala nola, masa eta inertzia-matrizea.

- **Visual:** *link*-aren propietate bisualak definituko ditugu hemen: hala nola, forma geometrikoa eta materiala.

- **Collision:** azkenik, oztopoen detekziorako motor fisikoak erabiliko duen kolisio eredua ezarriko dugu. Hau ez da nahitaez `Visual` elementuaren berdina izango, sinpleagoa izan daiteke konputazio-denbora optimizatzeko.

3.5 irudian ikus dezakegu *link* elementu bat.



Irudia 3.5: Link elementuaren adibide grafikoa.

Joint elementua

Joint elementua artikulazioaren zinematika eta dinamika definitzeko erabiltzen da, baita ere artikulazioaren mugak non dauden definitzeko.

Lehenik eta behin, artikulazio mota definitu beharko dugu atributu bezala. Horretarako, hurrengo 6 artikulazio motetatik bat aukeratu beharko dugu:

- **Revolute:** ardatz baten inguruan biratzen du eta goi eta behe mugak ezarrita ditu. Norabideak ezartzeko erabili daiteke.

- **Continuous:** aurrekoaren antzekoa baina mugarik gabe. Gurpiletan erabili daiteke.

- **Prismatic:** ardatza batean irristaka mugitzen den artikulazioa, goi eta behe mugak dituelarik.

- **Fixed:** hau ez da artikulazio bat, ezin baita mugitu. Hainbat kasutan erabilgarri izan daiteke, adibidez `base_laser` eta `base_link` *link*-ak lotzeko.
- **Floating:** artikulazio honek 6 askatasun graduko mugimendua ahalbidetzen du.
- **Planar:** artikulazio honek ardatzarekiko perpendikularra den planoan mugitzea ahalbidetzen du.

Behin artikulazio mota aukeratuta, hainbat elementu ditugu artikulazio hori gure beharren arabera konfiguratzeko, beharrezkoak soilik **arbasoa** eta **umea** direlarik (*revolute* eta *prismatic* motako artikulazioetan **limit** elementua ere beharrezkoa da). Ondoren, beste elementu batzuk erabili ditzakegu, adibidez:

- **Axis:** artikulazioan erabili beharreko ardatza. *Revolute* motako jointentzat errotazio-ardatza, *prismatic* motako ardatzentzat translazio-ardatza, eta *planar* motako jointentzat planoaren normala izango direnak. (x, y, z) moduko bektore normalizatu bezala ezarriko da.
- **Dynamics:** artikulazioaren propietate fisikoak ezartzeko balio du. Leuntzea eta marruskadura ezarri daitezke.
- **Limit:** lehen esan dugun bezala, elementu hau *revolute* eta *prismatic* motako artikulazioetan soilik erabiliko da. Goi eta behe mugez aparte, artikulazioari eragin ahal zaizkion esfortzu maximoa eta abiadura maximoa ezarriko ditugu.

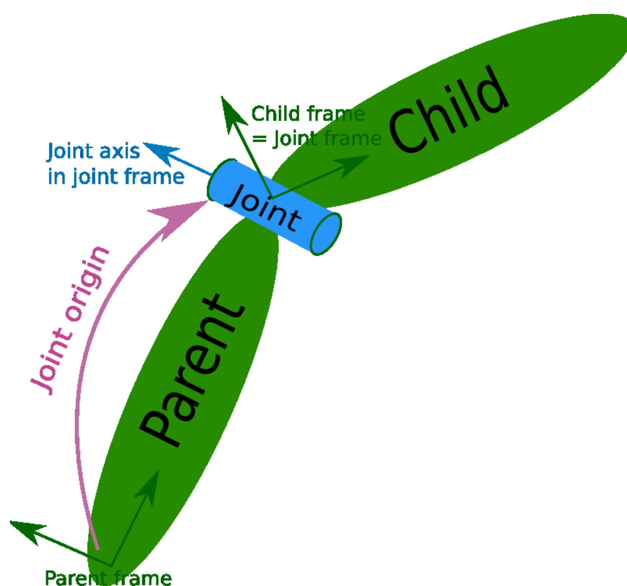
3.6 irudian ikusi dezakegu joint baten adibidea.

3.6 Simulazioa

Simulagailu robotiko bat fisikoki makina atzigarri egon gabe robot-softwarea garatzeko erabiltzen da. Soluzio honek erabilitako denbora eta kostua gutxitu ditzake. Kasu batzuetan, garatutako softwarea benetazko robotera transferitu daiteke aldaketa gutxirekin.

Ordenagailu bidezko simulazioa baliozko hautabidea izan daiteke kostu txikiko irtenbide bat nahi denean. Gainera, ordenagailu bidezko simulazioa erakargarri bihurtzen duten beste arrazoi batzuk izan ditzakegu kontutan:

- **Mugikortasuna:** edozein ingurune simulatu dezakegu bertan fisikoki egon gabe. Adibidez, proiektu honetan bezala, zentral termosolar bat simulatu dezakegu bertan egon gabe.



Irudia 3.6: *Joint* elementuaren adibide grafikoa.

- **Moldagarritasuna:** robotari edo inguruneari egiten zaizkion aldaketek ez dute kostu ekonomikorik.
- **Baliabideen kudeaketa:** simulagailu bat eta hardwarearen gainetik dagoen geruza bat erabiltzen direnean, robota atzigarri egon gabe garatu dezakegu softwarea. Hau bereziki baliagarria da hainbat pertsonak robot batekin lan egiten dutenean une berean.
- **Denboraren kontrola:** simulagailu batzuk denbora gelditzeko edo azkartzeko ahalmena dute. Hau baliagarria izan daiteke une zehatz bateko robotaren egoera aztertzeko, edo robotak denboran zehar duen portaera aztertzeko.

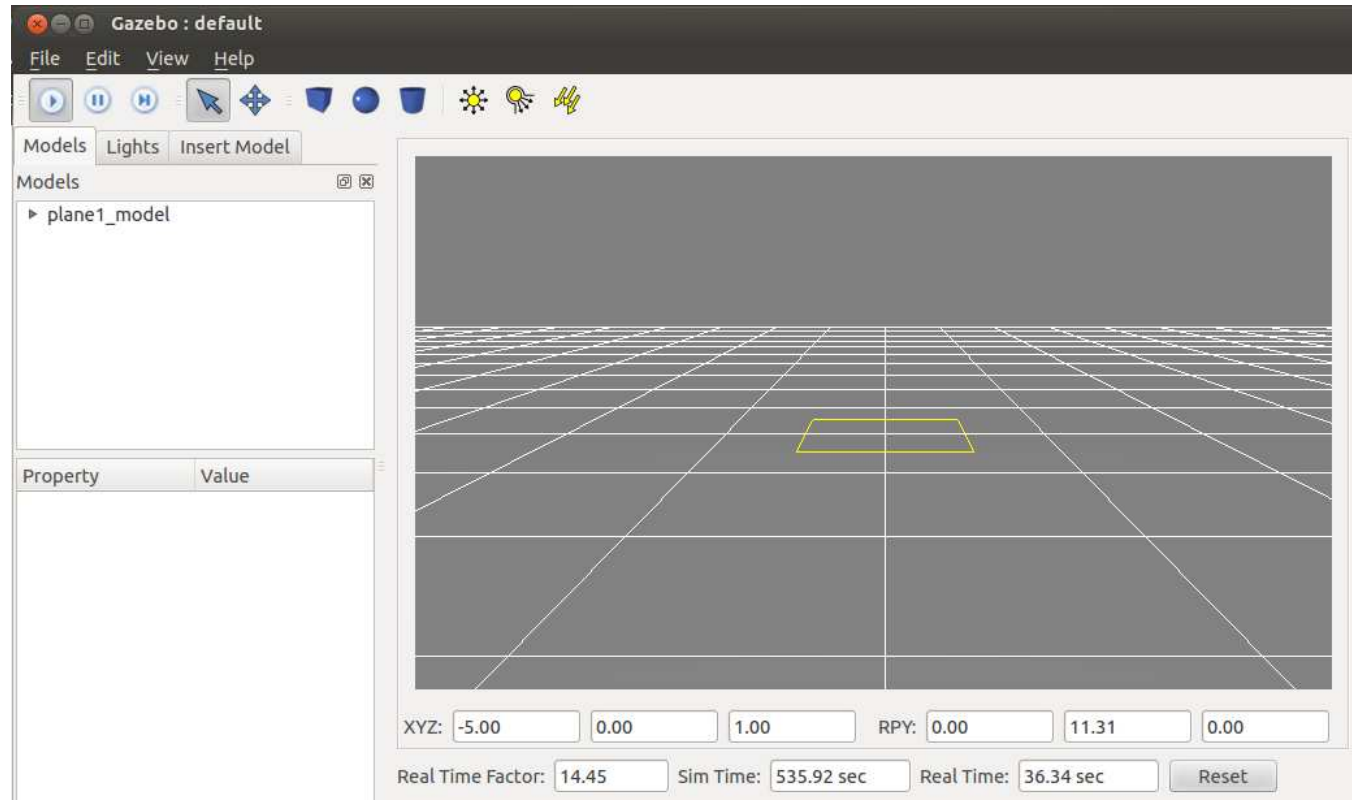
Hainbat simulagailu daude hautagai gaur egun, horien artean: Gazebo, Player/Stage, Carmen, Xraptor, RobotStudio, RobotWorks... Proiektu honetan Gazebo simulagailua erabili dugu, ROSekin batera erabiltzeko pentsatua dagoelako nagusiki.

3.6.1 Gazebo

Gazebo kanpo ingurunerako robot anitzen simulagailua da. Stage (Player proiektuaren zati) bezala, roboten populazio bat, sentsoreak edota objektuak simulatzeko gai da, baina hiru dimentsiotan. Sentsoreen berrelikadura

errealista eta objektuen arteko interakzio sinesgarriak sortzen ditu (objektu zurrunen simulazio fisiko zehatza du). Proiektu honen garapenerako Gazeboren 1.0.2 bertsioa erabili da.

Gazebo plataforma robotikoentzako algoritmoen garapen prozesuan laguntzeko garatu zen. Robotak eta inguruneak errealitatearen antzera simulatuz robot fisiko bat operatzeko idatzitako kodea bertsio artifizial batean exekutatu daiteke. Honela, bateria-galera arazoak, hardwarearen akatsak, robotaren jokabide arraroak... ekidin daitezke. Gainera, azkarragoa da simulazio ingurune bat abiatzea kodea robot errealean exekutatzea baino, bereziki simulazio inguruneak denbora erreala baina azkarrago lan egiteko gaitasuna badu. 3.7 irudian ikus dezakegu mundu huts bat simulagailuan.



Irudia 3.7: Mundu hutsa Gazebo simulagailuan.

4 Kapitulua

Ingurunea

Gaien Aurkibidea

4.1 Sarrera 38

4.2 Ingurunearen URDF eredua 38

4.1 Sarrera

Mainbot robotaren helburua mundu errealeko zentral termosolar batean mantenimendu operazioetan lagungarri izatea da. Arrazoi operatiboak direla medio, zentral errealean lan egitea ezinezkoa da, eta, horregatik, hautabide bezala zentral hori simulatzea erabaki da.

Zentral termosolarraren helburua eguzki-energia energia elektriko bilakatzea da. Horretarako, tutu baten barnean doan olio berezi bat berotzen da. Tutu horretan ahalik eta eguzki-izpi gehien konzentratzeko forma berezi bateko ispilu bat jartzen zaio inguruan.

Zentrala kolektoreen (*solar collector assembly*: SCA) unitateez osatua dago eta, era berean, unitatek hauek 12 kolektorez (solar collector element: SCE) osatuta daude. SCE bakoitza 28 ispiluz osatuta dago: 7 ispiluz osatutako 4 errenkada.

Kolektore-unitateak binaka daude kokatuta, iparretik hegoaldera orientatutako errenkada paraleloetan. Bi alboko errenkadek lau kolektoreko begizta bat osatzen dute, bukaeretan tutu batez elkartuta daudelarik.

4.2 Ingurunearen URDF eredua

Zentral termosolarra simulatzeko URDF eredu bat eraiki da hainbat `.STL` fitxategietatik abiatuta. Zentral guztia simulatu beharrean, U moduko kolektore begizta bakar bat simulatu da, gainerako begiztak honen berdinak direlako. Zehazki, ilara bakoitzak 300 metroko luzera izango du.

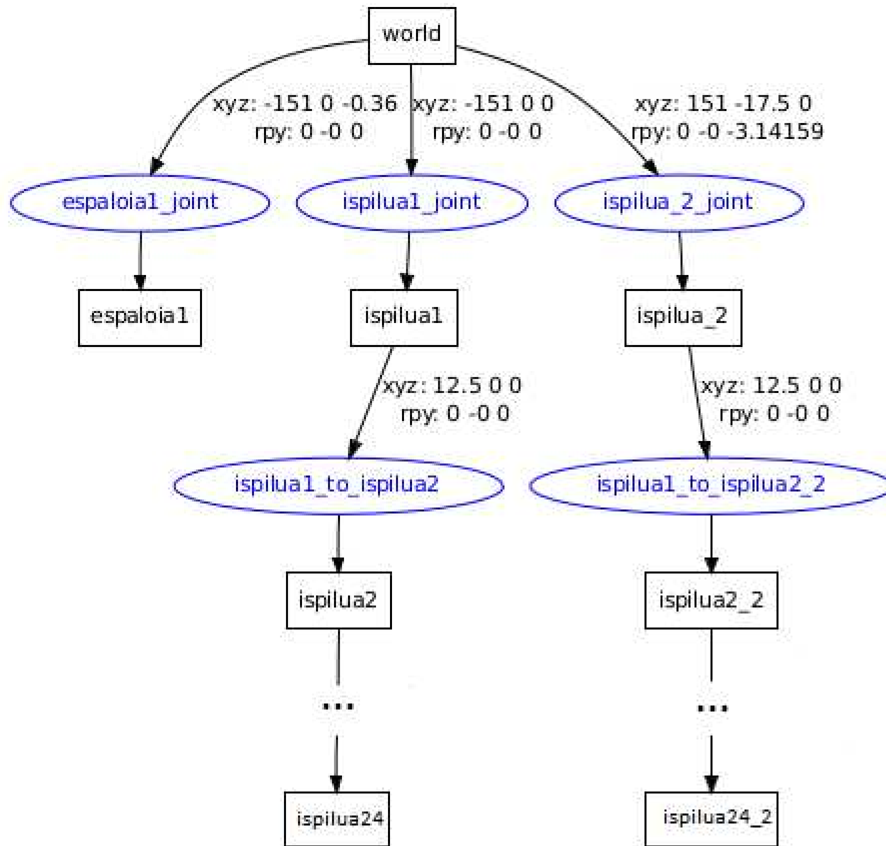
Zentralaren dimentsioak kontutan hartuta, eta kolektore bakoitzarentzako `link` bat erabiliz, hauen bi ilara egin ditugu, bata besteari begira. Horrela, ilara bakoitzean 24 kolektore jarri ditugu segidan.

Kolektore bakoitzaren **visual** eta **collision** elementuetarako ingurunea 3 dimentsiotan adierazten duen `.STL` fitxategi bat erabili da. Horrela, nahiz eta normalean *collision* elementuan *visual* elementuan baina bereizmen txikiagoa duen eredu bat erabiltzen den, kolisio-detekzioa behar denerako zehaztasun handia izango dugu.

Ondoren, espaloia ezarri dugu jarri beharreko tokian. Espaloi honetatik robota mugituko da, eta bere operazio-esparrua mugatzen du, ezin baitu hemendik atera.

URDFaren egitura orokorra 4.1 irudian ikusi dezakegu.

URDF fitxategia exekutatu ahal izateko, `mainbot_description` paketearen barruko `/urdf` azpidirektorioan sartu da berau. Exekutatzeko `launch` fitxategi bat osatu da `torresol.launch` izenarekin. Hona hemen bere edukia:



Irudia 4.1: Torresol zentralaren kolektore-segida.

```
1  <launch>
2
3    <!-- Denbora simulatua -->
4    <param name="/use_sim_time" value="true" />
5
6    <!-- Mundu hutsa hasieratzen du -->
7    <node name="gazebo" pkg="gazebo" type="gazebo" args
        ="$(find mainbot_description)/worlds/empty.world
        " respawn="false" output="screen"/>
8
9    <!-- Gazebo GUIa hasieratu -->
10   <node name="gazebo_gui" pkg="gazebo" type="gui"
```

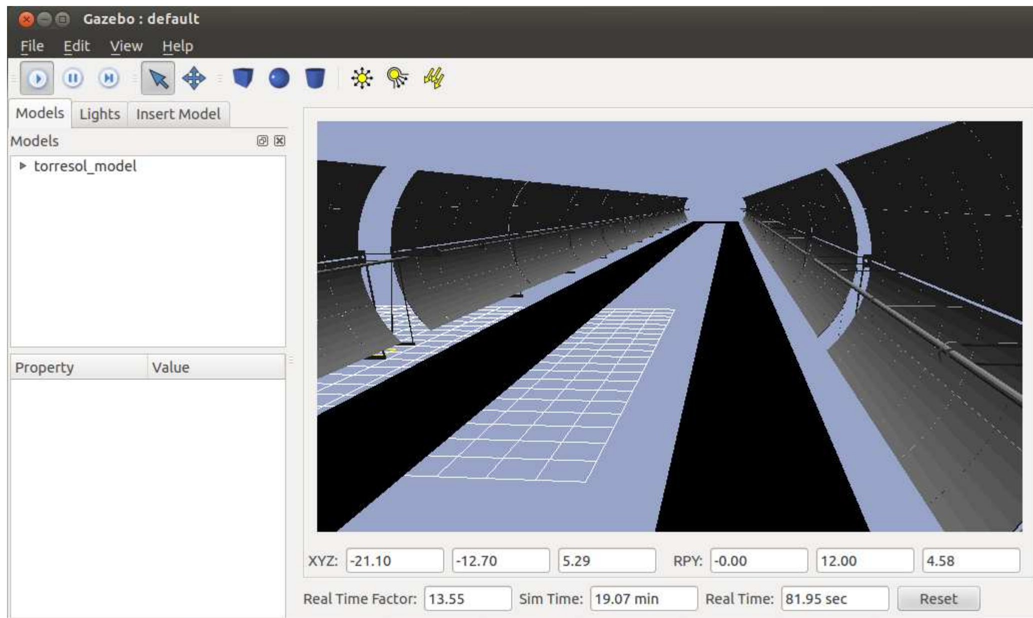
```
        respawn="false" output="screen"/>
11
12    <!-- Torresol URDF eredua parametro zerbitzarira
        bidali -->
13    <param name="world_description" textfile="$(find
        mainbot_description)/urdf/torresol.urdf" />
14
15    <!-- Parametro zerbitzaritik Torresol modeloa
        kargatu -->
16    <node name="spawn_torresol" pkg="gazebo" type="
        spawn_model" args="-urdf -param
        world_description -model torresol_model" respawn
        ="false" output="screen" />
17
18
19 </launch>
```

Launch fitxategia hasieratzeko ondorengo komandoa exekutatuko dugu:

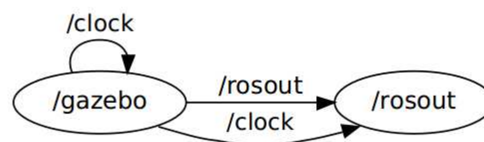
`roslaunch mainbot_description torresol.launch`

Dena ondo joan ezkero Gazebo irekiko da zentralaren URDF eredua kargatua duena. Probatutako sistemetan ez da inolako konputazio arazorik izan, normalean `Real Time Factor` altua lortzen delarik (15 inguru). Azken termino hau simulazioko denbora denbora-errelarekin konparatzen duen faktorea da. Adibidez, 1 faktorea izango bagenu denbora-errealean simulatzen ari garela esan dezakegu, eta, faktorea, handiagoa bada denbora-erreala baino azkarrago eta alderantziz. 4.2 irudian ikus dezakegu ingurune simulatua.

Azkenik, 4.3 ingurune simulatua abiatu ondoren martxan egongo diren nodoak ikus ditzakegu. Soilik Gazebo simulagailuari dagokion nodoa eta */rosout* izeneko nodo laguntzailea izango ditugu. Azken hau mezuak pantailaratzeko erabiltzen da nagusiki.



Irudia 4.2: Zentralaren simulazioa Gazebon.



Irudia 4.3: Ingurune simulatuaren exekuzio garaiko nodoak.

5 Kapitulua

Mainbot robotaren simulazioa

Gaien Aurkibidea

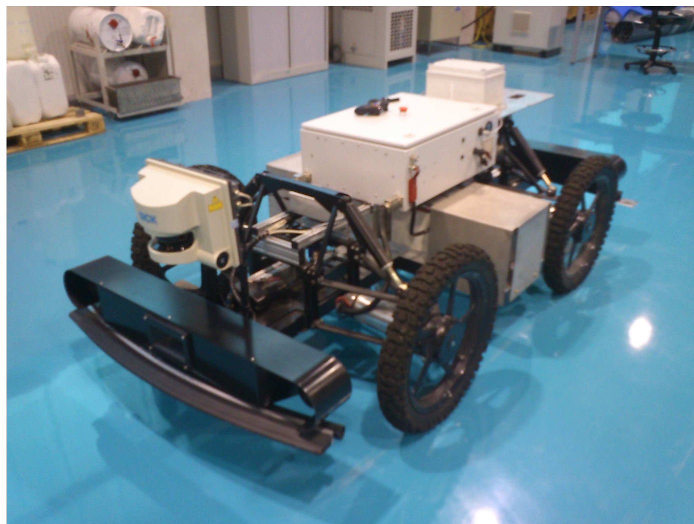
5.1 Mainbot robotaren deskribapena 44
5.1.1 Mainbot robot erreala 44
5.1.2 Robotaren URDF eredua 47
5.1.3 Pluginak 49
5.2 ROSeko pr2_mechanism pila 53
5.2.1 Pr2_controller_interface paketea 54
5.2.2 Pr2_controller_manager paketea 56
5.2.3 Pr2_mechanism_model paketea 56
5.3 Robotaren kontroladorea 59
5.3.1 Ackermann mugimendu-sistema 59
5.3.2 PID kontrola 60
5.3.3 Kontroladorea idazten 63
5.3.4 Gazebo Controller Manager plugina 68
5.4 Exekuzioa 68

5.1 Mainbot robotaren deskribapena

5.1.1 Mainbot robot erreala

Atal honetan proiektuan erabilitako robot mugikorra aztertuko dugu, Robosoft enpresak eraikitako Mainbot robota. Mainbot plataforma robotikoa ingurune zabaletan ibiltzeko diseinatua dago eta Ackermann norabide-geometria du. Hainbat sentsore ditu ingurunearen informazioa jasotzeko: laserra, ultrasoinu sentsoreak eta talka sentsoreak. Odometriarako, aldiz, dGPS sistema bat du, hau da GPS arrunt bat baliza lagungarri batekinbatera. Gure garapenean laser sentsorea soilik erabili dugu ingurunearen informazio nahikoa ematen digulako.

5.1 irudian ikus dezakegu mainbot robot erreala.



Irudia 5.1: Mainbot robot mugikorra.

Ezaugarri orokorrak

Lurreko robot hau Robosoften robuCarTT plataforman oinarrituta dago. Karga erabilgarri maximoa 300 kg-koa da eta lurzoru pabimentatuan zein malkartsuan ibiltzeko gai da. Bere ezaugarri teknikoak 5.1 taulan ikus ditzakegu.

Mainbot robotaren ezaugarriak	
Lurzoru mota	hirikoa, lur zatiekin
Oztopoak	250 mm
Lurrarekiko distantzia	280 mm
Aldapa	16º kargarekin/45º kargarik gabe
Zinematika	4 gurpil independente
	2 gurpil ardatz bakoitzeko
	2 ardatz mugikor
Pisua	350 kg (kargarik gabe)
Karga maximoa	300 kg
Abiadura maximoa	10 km/h
Neurriak	2300 x 1365 x 740 mm
Autonomia	3-4 h (erabileraren arabera)
Tentsio nagusia	45-54 V
Babesa	IP65 motoreen eta
	norabide mekanismoentzat
	IP54 zirkuitu elektronikoentzat
Hezetasuna	%0-90 kondentsaziorik gabe
Tenperatura (operazioan)	0...50ºC
Tenperatura (gordeta)	0...50ºC
Txertatutako kontrolagailua	Emtrion HICO 77-80
Programazio softwarea	RobuBOX

Taula 5.1: Mainbot robotaren ezaugarri teknikoak.

Laserra

Mainbot robotak nabigaziorako dituen sentsoreen artean LMS221 laserra aurkitu dezakegu (5.2 irudia). Laser hau robotaren aurrekaldean kokatuta dago

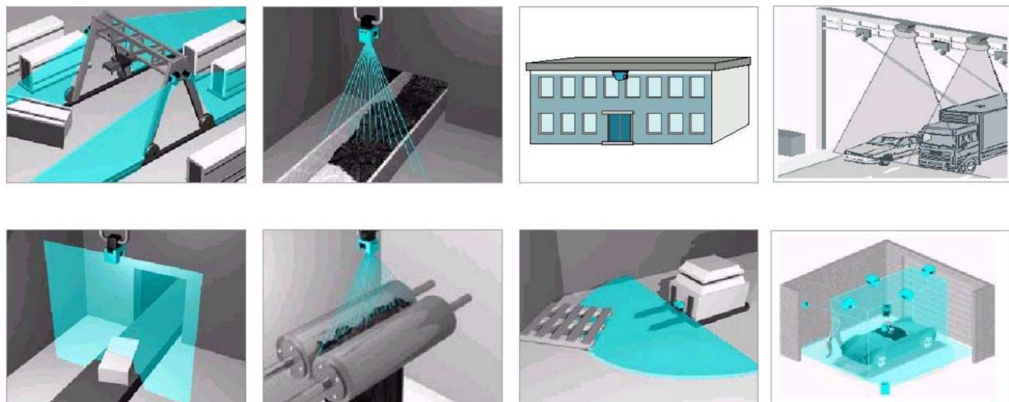
Laser hau hainbat aplikaziotan erabili daiteke, horien artean:

- Objektuen bolumena zehazteko (paketeak, paletak, kontainerrak... neurtzea).
- Objektuen posizioa zehazteko (paletak, kontainerrak, kontainer-ontziak...).
- Garabien edo, orokorrean, ibilgailuen talken detekziorako.

Irudia 5.2: LMS221 laser sentsorea.

- Atrakatze-prozesuen kontrolerako.
- Objektuen sailkapenerako (ibilgailuen detekzioa...).
- Prozesuen automazizaziorako (arrabolezko prentsak).
- Eraikinen segurtasuna bermatzeko espazio irekiak zaintzeko.

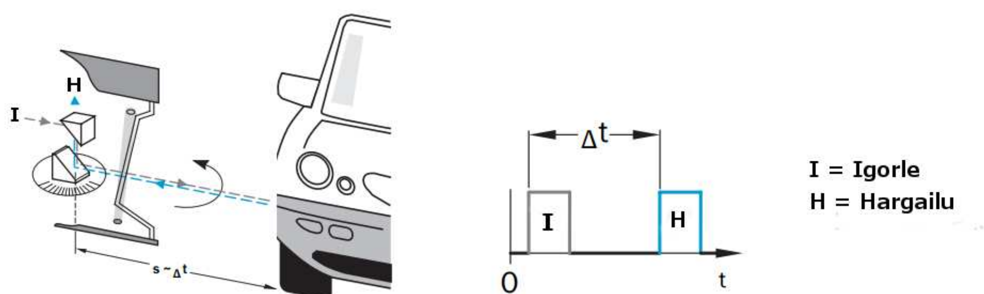
5.3 irudian ikus ditzakegu adibide hauek grafikoki.



Irudia 5.3: Laser sentsorearen zenbait aplikazio praktiko.

Gure kasuan, robotaren inguruneari buruzko informazioa jasotzeko erabiliko dugu, eta behar denean oztopoak ekiditen lagunduko digu. Ondoren, sakonki aztertuko ditugu bere ezaugarriak eta eragiketa-modua.

Eragiketa-printzipioa LMS221 laserrak laser argi-pultsuen hegaldi denbora neurtuz egiten du lan: laser-sorta pultsatua igortzen da eta bidean objekturik aurkitzen badu islaturik itzultzen da (ikus 5.4 irudia). Izpiaren islada LMS221 laserraren hargailuak erregistratzen du. Pultsuren igorpenaren eta jasotzearen arteko denbora laserraren eta objektuaren arteko distantziarekiko zuzenki proportzionala da.



Irudia 5.4: Laser sentsorearen eragiketa-printzipioa.

Laser-sorta pultsatua biratzeko gai den ispilu batek hainbat norabidetan igortzen du, eta, horrela, haizagailu baten antzeko forma duen laser ekorketa bat lortuko dugu laserraren ingurunean. Helburu-objektuen ingerada jasotako irakurketetatik abiatuz lortuko dugu. Neurketa datuak denbora errealean erabili daitezke erabiltzaileak zehaztutako atazetan, eta datuak trukatzeko interfaze bat dugu eskuragarri horretarako.

Behelainoaren zuzenketa automatikoa du laserrak aire zabaleko erabilerarako. Ur-tantak eta elur-malutak ekorketatik kentzen dira pixeletara zuzendutako ebaluazioaren bidez.

Ezaugarri teknikoak Laserraren ezaugarri teknikoak 5.2 taulan ikus ditzakegu.

5.1.2 Robotaren URDF eredua

Robotaren URDF eredua oso sinplea da, 11 *link*-ez eta hauek elkarren artean erlazionatzen dituzten 10 *joint*-ez osatua dago.

Irudia 5.5: LMS221 laserraren transmisioaren norabidea.

LMS221 laser sentsorearen ezaugarriak	
Orokorra	
Barruti maximoa	80 m
Bereizmen angeluarra	0.25º/0.5º/1.0º (aukeran)
Erantzun-denbora	53/26/13 ms
Neurketaren bereizmena	10 mm
Elektrikoa	
Datuen interfazea	RS 232/RS 422 (aukeran)
Transferentzia-abiadura	9.6/19.2/38.4/500 kBd
Elikatze tensioa (elektronika)	24 V DC ± %15
Elikatze tensioa (berogailua)	24 V DC
Energia erabilera	$\tilde{2}$0 W
Giroko tenperatura (operazioan)	-30...50ºC
Mekanikoa	
Itxituraren ebaluazioa	IP 67
Pisua	$\tilde{9}$ kg
Dimentsioak	185 x 156 x 210 mm;
	kableekin: 185 x 156 x 265 mm

Taula 5.2: Laser sentsorearen ezaugarri teknikoak.

Lehenik, robotaren oinarriaren *linka* dugu, `base_link` izenekoa. Honek 400 kg-ko masa du eta inertzia balioak SolidWorks[1] programatik atera dira.

[1]Ikus http://www.solidworks.com webgunea.

Aipatu beharra dago **yaw** balioa pi/2 radianetan jarri dela robota zentralaren orientaziora egokitzeko. **visual** eta **collision** elementuei dagokienez, aipatutako orientazio egokitzapenaz aparte, filtro baten bidez murriztutako `.STL` fitxategi bat erabili da. Hau da, jasotako `.STL` fitxategia erabili ordez MeshLab [2] programaren laguntzaz honen konplexutasuna jaitsi da konputazioa arintzeko.

Robota mugitzeko 4 gurpil ditugu bakoitza bere *link*-arekin, eta gurpil bakoitzeko beste *link* bat dago gurpil horren motorra adierazteko. Azken *link* hau izango da gurpilaren norabidea ezarriko duena. Hemen, aipatu beharra dago `base_link` *link*-etik motorraren *link*-era doan *joint*-a **revolute** motakoa dela, hau baita norabideak aldatzeko *joint* mota egokia. Motorraren *link*-etik gurpilaren *link*-era doan *joint*-a, aldiz, **continuous** motakoa da, hau delako mugarik gabe bira dezakeen gurpil batentzat *joint* mota egokia.

Gainera, aurrerago azalduko dugun laser sentsorearentzako beste *link* bat ezarriko dugu robotaren aurrekaldean. *Link* honi aurrerago laser *plugin* bat gehituko diogu laser irakurketak *link* honi lotuak egon daitezen. *Link* honi forma emateko `box` motako elementu bat erabili da **visual** eta **collision** elementuetan.

Azkenik, oinarriaren linkaren azpitik `base_footprint` izeneko *link* lagungarri bat jarriko dugu. Aurrerago ikusiko dugun moduan, *link* hau `navi-gation` paketea erabili ahal izateko baldintza bat da.

5.6 irudian ikusi dezakegu URDF fitxategiaren egitura orokorra.

5.1.3 Pluginak

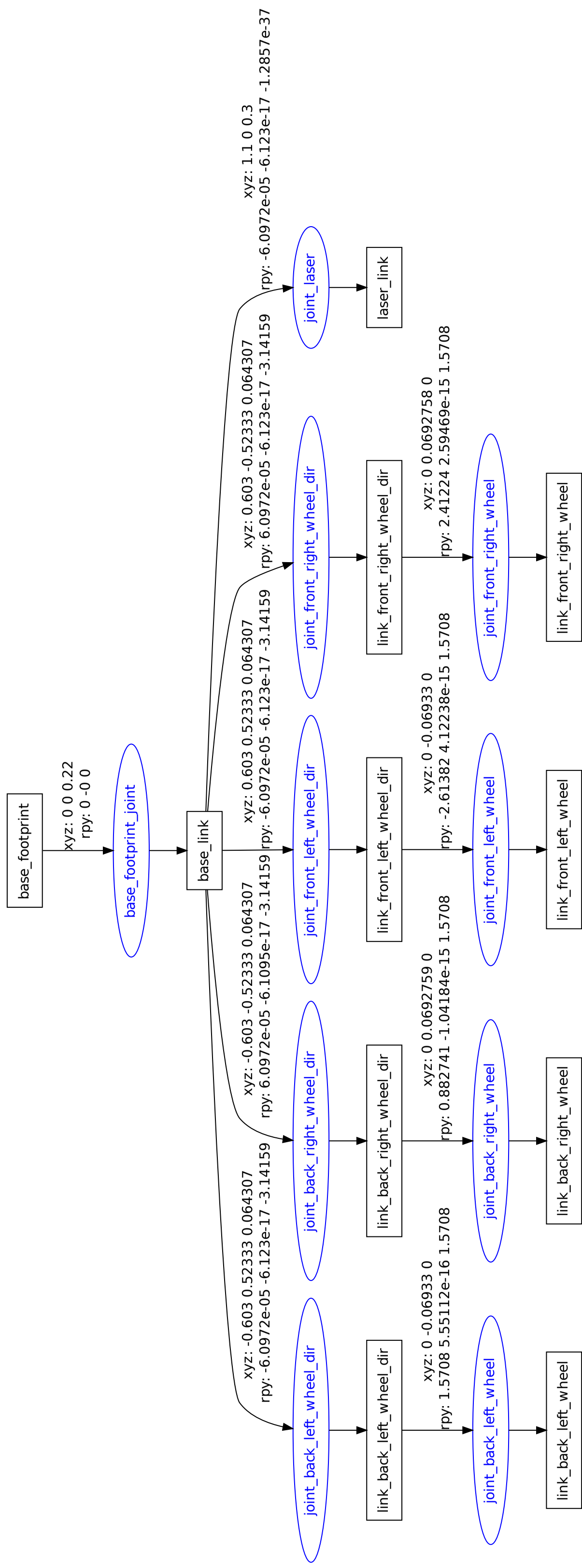
Robota simulatzeko orduan, robotaren parte zurrunak ez ezik, robotak dituen sentsoreak simulatzeko mekanismo bat behar dugu simulazio errealista bat lortu nahi badugu. Gure robotean, laser sentsorea eta GPS sentsorea simulatu behar ditugu, hauek sistemaren atal ezinbestekoa baitira. Horretarako, Gazebon *plugin* bezala ezagutzen den mekanismoaz baliatuko gara.

Gazeboko *plugin* bat liburutegi partekatu bat bezala konpilatzen den eta, ondoren, simulazioari gehitzen zaion kode zati bat da. Pluginak Gazeboren funtzionalitate guztia du atzigarri C++ klase estandarren bidez.

Laser sentsorearen ***plugin***-a

Lehenago esan bezala, gure robotak Sick LMS 221 motako laserra dauka muntaturik aurrekaldean. Sentsore hau nabigaziorako eta oztopoak ekiditeko erabiliko dugu eta, beraz, gure sistemaren atal kritikoa da.

[2] Ikus http://meshlab.sourceforge.net webgunea.



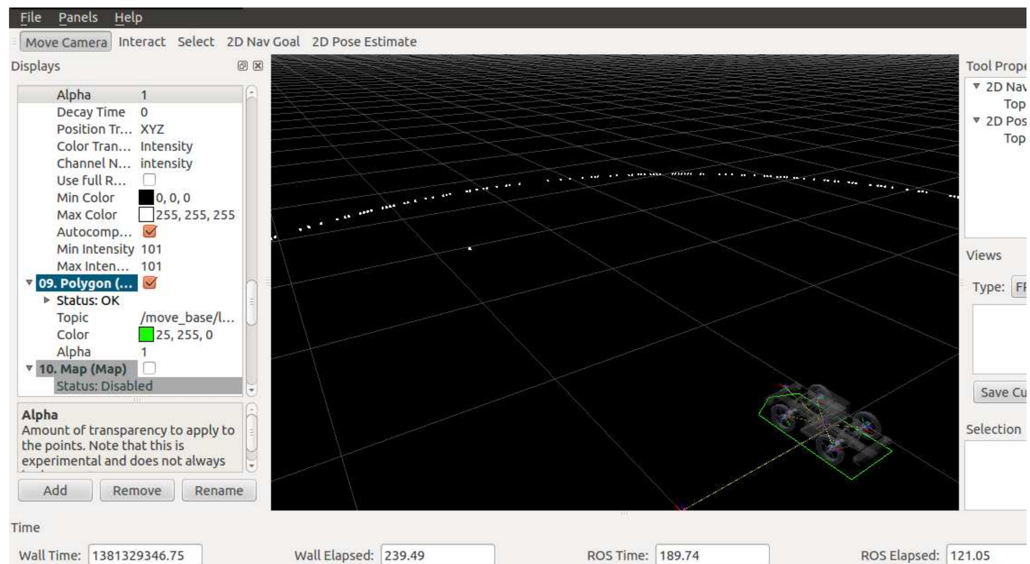
Irudia 5.6: Mainbot robotaren URDF eredua.

Laserra Gazebon simulatzeko laserraren plugina erabiliko dugu. Horretarako, aurretik ikusi dugun `laser_link` *link*-a erabiliko dugu laserraren *link* bezala eta, honi, *plugin*-a atxikituko diogu. *Plugin* hau gure benetazko laserrak dituen ezaugarriak kontutan hartuz konfiguratuko dugu. Jakinda LMS 221ak 180°ko ikusmen-eremua izango duela, eta 0.5° bereizmen-angeluarra erabiliko dugula, 360 izpi simulatzeko agindua emango diogu pluginari.

Ondorengo kodean ikusi dezakegu gure pluginaren konfigurazio osoa.

```
1
2   <gazebo reference ="laser_link">
3     <sensor:ray name="sick_laser">
4        <rayCount>360</rayCount>
5        <rangeCount>360</rangeCount>
6        <laserCount>1</laserCount>
7
8        <origin>0.0 0.0 0.0</origin>
9        <displayRays>fan</displayRays>
10
11       <minAngle>-90.0</minAngle>
12       <maxAngle>90.0</maxAngle>
13
14       <minRange>1</minRange>
15       <maxRange>20.0</maxRange>
16       <resRange>0.01</resRange>
17       <updateRate>10.0</updateRate>
18       <controller:gazebo_ros_laser name="
            gazebo_ros_sick_laser_controller" plugin="
            libgazebo_ros_laser.so">
19         <gaussianNoise>0.005</gaussianNoise>
20         <alwaysOn>true</alwaysOn>
21         <updateRate>10.0</updateRate>
22         <topicName>sick_laser_topic</topicName>
23         <frameName>laser_link</frameName>
24         <interface:laser name="
              gazebo_ros_sick_laser_iface" />
25       </controller:gazebo_ros_laser>
26    </sensor:ray>
27   </gazebo>
```

Azkenik, 5.7 irudian laserraren irakurketak RViz bistaratze-tresnan ikusi ditzakegu.



Irudia 5.7: Laser sentsorearen irakurketak RViz-en.

GPS sentsorearen plugina

Gure roboteko GPS sentsorea simulatzeko P3D pluginaz baliatuko gara. Sentsore honek simulazioko objektu baten posizioa eta abiadura publikatzen ditu `nav_msgs::Odometry` motako mezuen bidez. Mota honen osagaiak ondorengoak dira:

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose //posizioa
geometry_msgs/TwistWithCovariance twist //abiadura
```

Plugin hau gure URDF fitxategian txertatzeko ondorengo kode zatia gehitu dugu:

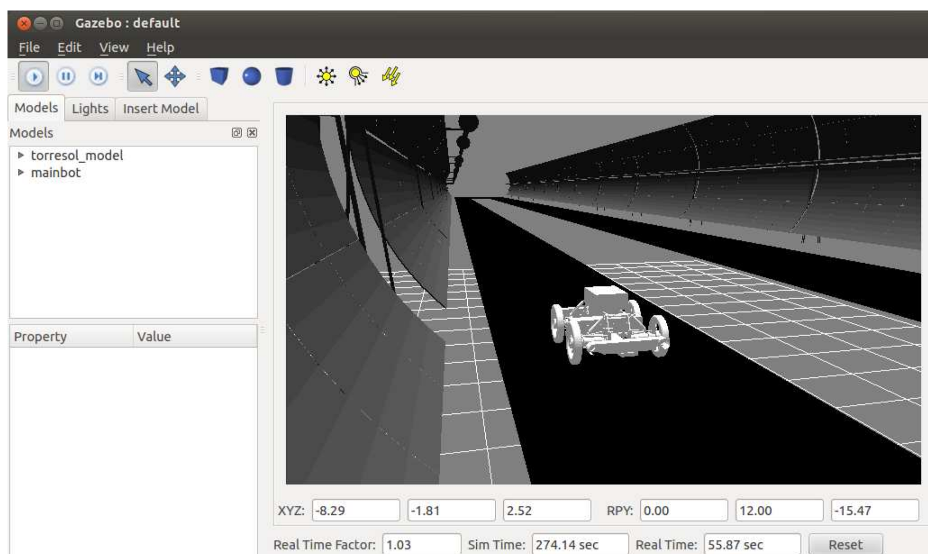
```
1  <gazebo>
2    <controller:gazebo_ros_p3d name="
       p3d_base_controller" plugin="libgazebo_ros_p3d
       .so">
3      <alwaysOn>true</alwaysOn>
4      <updateRate>100.0</updateRate>
5      <bodyName>base_link</bodyName>
6      <topicName>/base_ground_truth</topicName>
7      <gaussianNoise>0.01</gaussianNoise>
```

```
8        <frameName>map</frameName>
9        <rpyOffset>0.0 0.0 0.0</rpyOffset>
10       <interface:position name="p3d_base_position" />
11     </controller:gazebo_ros_p3d>
12   </gazebo>
```

Honen arabera, `/base_ground_truth` topic-ean `nav_msgs::Odometry` motako mezuak publikatuko dira 100 Hz-eko frekuentziarekin. Mezu horietako bakoitzean `/map` framearekiko `/base_link` framearen posizioa eta abiadura publikatuko dira, zarata pixka bat gehituz.

Aurrerago ikusiko dugun bezala, nabigazioa konfiguratzeko orduan robotaren posizioa soilik interesatuko zaigu.

Amaitzeko, Mainbot robot simulatua Gazebon ikus dezakegu 5.8 irudian.



Irudia 5.8: Mainbot robota Gazebo simulagailuan.

5.2 ROSeko pr2_mechanism pila

`Pr2_mechanism` pilak PR2 robota benetazko denbora errealeko kontrol begizta baten bidez kontrolatzeko beharrezko azpiegitura guztia du. Gure robota erabiltzeko pila hau beharrezkoa ez den arren, ROS programatzaileen artean, ohikoena, kontroladoreak idazteko pila hontaz baliatzea da, beti ere norberaren beharretara egokiturik.

Pila honen osagai nagusiak, eta guk erabiliko ditugunak ondorengoak dira:

- **pr2_controller_interface:** denbora errealeko kontroladore baten C++ interfazea dugu hemen.
- **pr2_controller_manager:** denbora errealeko begizta batean hainbat kontroladore exekutatu eta kudeatzeko balio duen azpiegitura.
- **pr2_mechanism_model:** urdf robot deskripzio formatuan oinarritutako robot bat esfortzuen bidez kontrolatzeko eredua.

5.2.1 Pr2_controller_interface paketea

Pakete honek denbora errealeko kontroladore baten interfazea definitzen du. Interfaze hau inplementatzen duen kontroladore bat **controller manager**-aren bitartez exekutatu daiteke denbora errealeko kontrol begiztan. Pakete honek kontroladore guztiek heredatu behar duten C++-eko kontroladore-klase oinarrizkoa du.

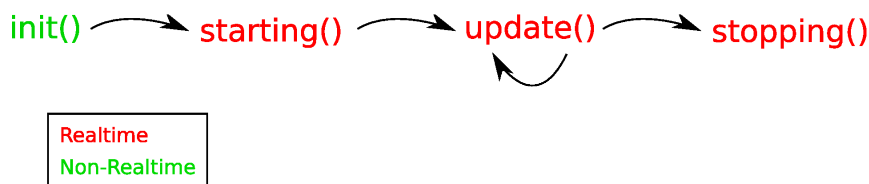
Esan bezala, denbora errealeko kontroladore bat inplementatzeko gure kontroladoreak `pr2_controller_interface::Controller` klasea heredatu behar du. Klase honek ondorengo ezaugarriak ditu:

- Inplementatu beharreko lau metodo: **init**, **starting**, **update** eta **stopping**.
- Dei daitekeen metodo bat: **getController**.

Hemen ikusi dezakegu klase honen kodea:

```
1  namespace pr2_controller_interface
2  {
3    class Controller
4    {
5    public:
6      virtual bool init(pr2_mechanism_model::RobotState
           *robot,
7                        ros::NodeHandle &n);
8      virtual void starting();
9      virtual void update();
10     virtual void stopping();
11
12     bool getController(const std::string& name,
13                        int sched,
14                        ControllerType*& c);
15   };
16 }
```

5.9 irudian ikusi dezakegu lau metodo hauek jarraitzen duten exekuzio-fluxua.



Irudia 5.9: Kontroladoreen metodoen exekuzio fluxua.

Pausoz pauso komentatuko ditugu metodo horiek:

Lehenik `init` metodoa exekutatuko da **denbora ez-errealean**. Metodo hau kontroladore bat kargatzen denean exekutatzen da, kontroladorea hasieratzeko. Aipatu beharra dago ez dela berdina kontroladore bat inizializatzea edo hasieratzea: inizializazioa hasieratzea baino nahi haina lehenago egin daiteke. `Init` metodoak bi argumentu hartzen ditu:

- **robot:** hau `pr2_mechanism_model::RobotState` bat da, robot eredua deskribatzen duena (pr2_mechanism_model-en zehaztua). Ereduak robotaren *joint*-ak (aktuadoreak, kodegailuak, etab.) atzitzea ahalbidetzen du eta robot mekanismoaren deskribapen zinematiko/dinamikoa du.

- **n:** hau `ros::NodeHandle` motakoa da eta kontroladorearen izen-espazioa da. Node maneiatzaile honen izen-espazioan, kontroladoreak parametro zerbitzaritik konfigurazioa irakurri dezake, topiak argitaratu, etab.

`Init` metodo honek inizializazioa arrakastatsua izan den ala ez itzuliko du. Inizializazioak porrot egiten badu, `pr_controller_managerr`-ak kontroladorea deskargatuko du. Kontroladore bat soilik behin inizializatu daiteke.

Kontroladorea inizializatu ondoren hasieratu egingo dugu. Hau benetazko denbora errealean egingo da. *Controller Manager*-a izango da hau egitearen arduradun eta kontroladorea exekutatzen den bakoitzeko behin deituko zaio. Metodo honen exekuzioa lehen eguneratze-deiaren ziklo berdinean egingo da, justu eguneratze-deia baina lehenago.

`Starting` metodoak kontroladorea inizializatuko du lehen eguneratze deia baina lehenago. *Controller Manager*-ak aurrerago nahi duen unean hasi dezake berriz kontroladorea, kontroladorea kargatzeko edo deskargatzeko beharrik gabe.

Hasieraketa honen ondoren, `update` metodoa exekutatuko da exekuzio ziklo bakoitzean eta benetako denbora errealean. Hau *Controller Manager*-ak egingo du 1000 Hz frekuentziarekin. Honen arabera, exekuzioan dauden kontroladore guztiek konbinatuta ezingo dute 1 ms baina luzeago igaro. Eguneratze-begiztan benetako kontrol-lana egiten da.

Azkenik, `stopping` metodoa deituko da benetazko denbora errealean kontroladorearen exekuzioa amaitzeko. Hau azken eguneratze-begiztaren ziklo berdinean egiten da, justu eguneratze-begiztaren ondoren.

5.2.2 Pr2_controller_manager paketea

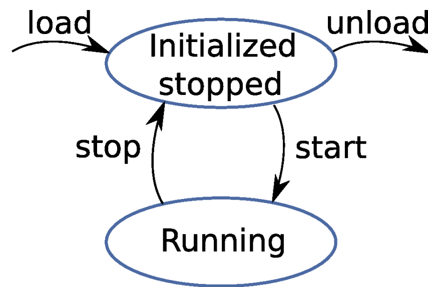
Controller manager (CM) paketeak benetazko denbora errealeko begizta batean kontroladoreak exekutatzeko azpiegitura ematen du (5.10 irudia). Kontrol-begiztaren ziklo bakoitzean CM barnean dauden kontroladore guztiak eragingo dira. Kontroladoreak zein ordenetan eragiten diren CM schedulerrak ezarriko du. Kontroladoreak kargatu/hasieratu/gelditu/deskargatzeko zerbitzuak eskaintzen ditu.

`Pr2_controller_manager`ak benetazko denbora errealko begizta bat eskaintzen du robot mekanismoa kontrolatzeko. Robot mekanismoa esfortzuen bidez kontrolatutako *joint* multzo batek osatzen du (`pr2_mechanism_model`-en azaldua). PR2 robotarentzat, 1000 Hz-tan exekutatzen da kontrol-begizta. *Controller Manager*-ak bakoitzaren denbora errealeko kontroladorea bere kontrol begiztan kargatzeko azpiegitura eskaintzen du. Kargatzen den kontroladore bakoitza milisegundoro deitua izango da. *Controller Manager*-ak segurtasun mugak ezartzen ditu, horrela *joint* batek jasan dezakeena baino esfortzu handiagoa jasateaz babestuz.

ROS sisteman *joint* guztien egoerak publikatzen ditu *Controller Manager*-ak, `sensor_msgs/JointState` motako mezuen bidez. Mezu hauek `joint_state` topic-ean argitaratzen dira, 100 Hz frekuentziarekin. Frekuentzia hau aldatzeko `joint_state_publish_rate` parametroa egokitu daiteke.

5.2.3 Pr2_mechanism_model paketea

Pakete honek *Controller Manager*-aren barnean erabiltzen diren denbora errealeko kontroladorek erabiltzen duten robot-eredua du. Robot-ereduak denbora errealeko begizta batean robot-mekanismoa kontrolatzean jartzen du arreta, eta horregatik denbora errealean garrantzitsuak diren robotaren osagaiak ditu soilik: robotaren *joint*-ak (kodegailuak, transmisioak eta eragingailuekin).



Irudia 5.10: Kontroladoreen egoera posibleak.

`Pr2_mechanism_model` paketeak `pr2_mechanism_model::RobotState` C++ klasea du, robot ereduaren deskribapena eta robotaren *joint*-en interfaze bat dena. `RobotState`-ak *joint* banakoetara sarbide erraza ahalbidetzen du.

5.11 irudian dagoen diagraman ikusi dezakegu robotean eta simulazioan nola erabiltzen den mekanismo eredua.

Pakete honen klase nagusia `pr2_mechanism_model::RobotState` da. Kontroladorea hasieratzen denean, *Controller Manager*-ak kontroladoreari `RobotState`-ra erakusle bat pasatzen dio (ikus kontroladorearen interfazea). Robot-egoerak (*RobotState*) robotaren eredu zinematiko/dinamikoa eta uneko egoera deskribatzen ditu. Robotaren egoera robotaren *joint*-en posizioa/abiadura/esfortzuek definitzen dute. Robotaren eredua URDF objektu bat da, URDF paketean definituta dagoen bezala. Gainera, `RobotSate`ak 'kontroladore denbora' ematen digu, kontroladore zikloa hasten den unea.

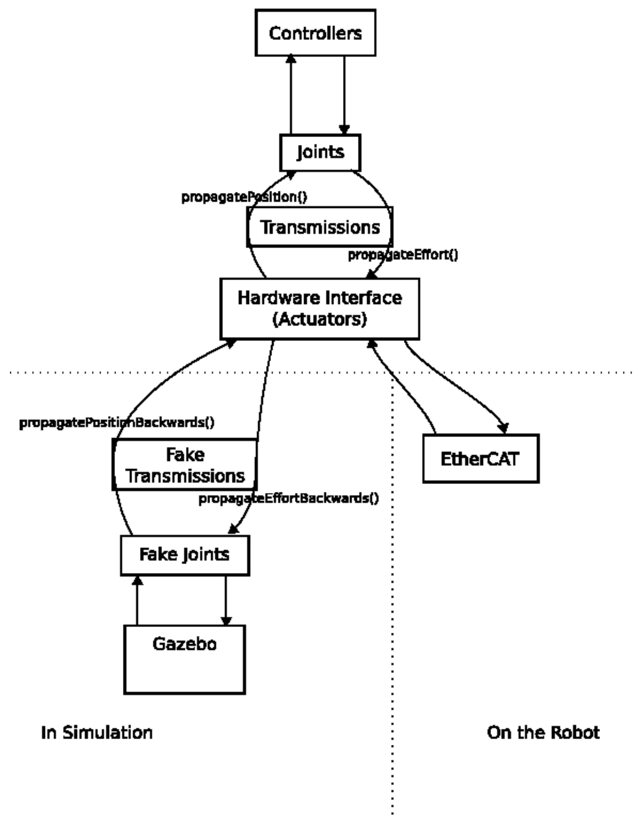
`RobotState`-aren osagaiak ondorengoak dira:

- **pr2_mechanism_model::JointState:**

  `pr2_mechanism_model::RobotState`-ak izen bidez bere barnean dauden `pr2_mechanism_model::JointState` guztiak atzitzea ahalbidetzen du. `JointState` bat atzitzeko, hurrengo metodoa erabili behar da:

  ```
  JointState* js = robot_state_->getJointSate
  (izena);
  ```

  `JointState` batetik *joint*-aren posizioa, esfortzua eta abiadura atera daitezke, baita aplikatu nahi den esfortzua bidali. Adibidez, `JointSta-te`-tik jointaren neurtutako posizioa atera nahi badugu, hurrengo kode-lerroa erabili dezakegu:



Irudia 5.11: Mekanismo ereduaren robotaren eta simulazioaren konfigurazioa.

```
double posizioa = js->position_;
```

Bestalde, esfortzua bat bidaltzeko `JointState`ari, hurrengo kodelerroa exekutatu dezakegu:

```
js->commanded_effort_ = my_command;
```

- **Transmisioak:** `pr2_mechanism_model::RobotState`-ak motorren eta *joint*- arteko transmisioak atzitzea ahalbidetzen du. Normalean ez dago transmisioak erabiltzeko beharrik, zinematika eredu aurreratu bat erabili ezean, edo *joint*-ak kalibratzen ari ez bagara. `Joint State` bat atzitzeko komando hau erabiliko dugu:

```
Transmission* tr = robot_state_->
getTransmission(izena);
```

- **Kontroladorearen denbora:** kontroladore ziklo bat hasten den ba-

koitzean, *Controller Manager*-ak denbora erregistratzen du. Denbora hau edozein kontroladorek atzitu dezake `pr2_mechanism_model::RobotState` bitartez. Kontroladore batek ondoz ondoko bi zikloren arteko denbora neurtu nahi duenean, denbora hau erabili beharko du, eta ez sistemaren denbora. Sistemaren denborarekin alderatuta, beste kontroladoreek eguneratze begiztan erabiltzen duten denborak ez du kontroladorearen denboran eraginik. Gainera, kontroladorean denbora da hardwarearekin komunikazioa dagoen unea neurtzeko modu egokiena. Kontroladorearen denbora atzitzeko hurrengo komandoa erabiliko dugu:

```
ros::Time denbora = robot_state_->getTime();
```

- **Robot eredua:** `pr2_mechanism_model::RobotState` barruan urdf robotaren deskribapen-objektu bat dago. URDF eredua atzitzeko kode-lerro hau erabiliko dugu:

```
urdf::Model m = robot_state_->robot_->
robot_model_;
```

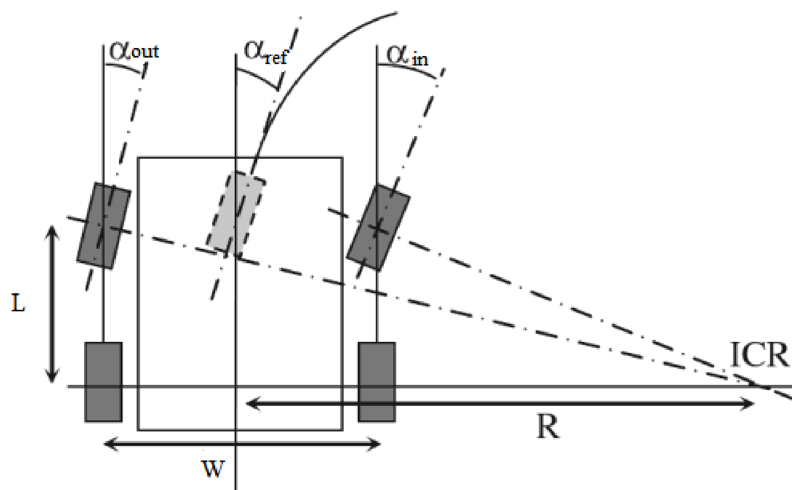
5.3 Robotaren kontroladorea

Kontroladorearen atzetik dagoen azpiegitura guztia ikusita, gure kontroladorea idazteko jarraitutako urratsak zehatz-mehatz deskribatuko ditugu. Lehenik, kontroladorearen idazketa deskribatuko dugu eta, ondoren, *joint*-ei gehitutako PID kontrola deskribatuko dugu.

5.3.1 Ackermann mugimendu-sistema

Ackermann mugimendu-sistema gurpilen neumatikoen higadura gutxitzeko sortutako norabide sistemaren konfigurazio mota bat da. Sistema honetan sistemaren gurpil guztiak **Instantaneous Center of Rotation** (ICR) izeneko puntu bat izango dute komunean.

Ackermann geometriaren intentzioa gurpilek kurbak jarraitzen dituztenean aldeetara irristatzea ekiditea da. Honetarako soluzio geometrikoa ondorengoa da: gurpil guztien ardatzak komunean duten puntu batekiko erradioarekin edukitzea. Atzeko gurpilak robotaren translazioaren noranzkoan soilik mugitzen direnez, puntu zentral hori atzeko ardatzak definitzen duen lerrozuzenean egon beharko da kokatuta nahitaez. Gainera, aurreko gurpilei dagokienez, bi hauen ardatzak atzeko ardatzaren lerrozuzenean puntu berdinean ebakidura izan dezaten, barruko gurpila kanpoko gurpila baino angelu txikiagoarekin biratu beharko da (5.14 irudia).



Irudia 5.12: Ackermann geometriaren adierazpide grafikoa.

Gure sisteman autoak biratzeko α_{ref} angelu bat izango dugunez, egin behar dugun aurreneko pausua atzeko ardatzaren erdigunetik ICC punturaino dagoen distantzia, R bezala ezagutua, kalkulatzea izango da. Horretarako, aurreko eta atzeko gurpilen arteko ardatzen distantzia L bezala jakina dugu, eta, ondorioz, 5.1 ekuazioa aplikatuta aterako dugu R balioa.

$$R = \frac{l}{\tan \alpha_{ref}} \tag{5.1}$$

ICC puntuaren erradioa kalkulatu dugula, erlazio trigonometrikoak aplikatuz α_{in} eta α_{out} angeluak aterako ditugu dagokion gurpilari bere norabidea esleitzeko (5.2 eta 5.3 ekuazioak).

$$\alpha_{in} = \arctan \frac{l}{R - w/2} \tag{5.2}$$

$$\alpha_{out} = \arctan \frac{l}{R + w/2} \tag{5.3}$$

5.3.2 PID kontrola

PID kontrol mekanismoa atzeraelikadura bidez lortu nahi den balio bat eta lortu den balio baten arteko desbiazioa kalkulatzeko erabiltzen da, eta, ondoren, prozesua doitu daiteke eragiketa zuzentzaile bati esker. PID kontrolaren

kalkulu-algoritmoak hiru osagai ditu: proportzionala, integrala eta deribatiboa.

Prozesu edo sistema bat erregulatzen duen PID kontroladore batek ondo funtzionatzeko ondorengo elementuak behar ditugu:

1. Sistemaren egoera determinatuko duen sentsore bat (termometroa, kodegailua...).
2. Eragingailua zuzenduko duen kontroladore bat.
3. Sistema modu kontrolatuan aldatuko duen eragingailu bat (motorra, erresistentzia elektrikoa, ...).

Proportzionala

Kontrol proportzionaleko algoritmoan, kontroladorearen irteera errorearekiko proportzionala da. Hau da, helburua prozesu aldagaia eta helburuaren arteko diferentzia da. Beste hitz batzutan esanda, kontroladore proportzional baten irteera errore seinale baten eta irabazi proportzional baten arteko biderkadura da.

Hau horrela idatzi daiteke matematikoki:

$$P_{out} = K_p e(t) \tag{5.4}$$

non

- P_{out}: kontroladore proportzionalaren irteera.
- K_p: irabazi proportzionala.
- $e(t)$: t unean prozesuaren errorea. $e(t) = SP - PV$.
- SP: helburua.
- PV: prozesu-aldagaia.

Integrala

Kontroladorearen termino Integralak termino Proportzionalak eragiten duen egoera gerakorreko errorea gutxitu edo ezabatzea du helburu. Kontrol integralak prozesu-aldagaiaren eta prozesu-helburuaren artean desbiazioak ematen direnean hartzen du parte, eta desbiazio hau denboran integratzen du ondoren ekintza proportzionalari gehitzeko. Errorea integratu egiten da, honen funtzioa errorea denbora periodo baten zehar erdibanatzea edo gehitzea

da. Ondoren, **I** konstante bat biderkatzen zaio. Azkenik, integralaren emaitza proportzionalari gehitzen zaio PI kontrola osatzeko, eta horrela egoera gerakorreko errorerik gabeko erantzuna izango du sistemak.

Integralaren formula ondorengoa da:

$$I_{out} = K_i \int_0^t e(\tau) \mathrm{d}\tau \tag{5.5}$$

non:

- I_{out}: kontroladore integralaren irteera.
- K_i: irabazi integrala.
- t: denbora.
- $e(t)$: t unean prozesuaren errorea. $e(t) = SP - PV$
- τ: integrazio aldagaia: 0 denboratik t unerako balioak hartzen ditu.

Deribatiboa

Termino deribatiboa errorearen balio absolutuan aldaketa bat dagoenean aktibatzen da. Ekintza deribatiboaren funtzioa errorea sortzen den abiadura berdinean proportzionalki zuzentzea da, errorea minimoan mantentzeko asmoz. Denborarekiko deribatzen da eta **D** konstante batekin biderkatzen da. Ondoren, aurreko seinaleei (P+I) gehitzen zaie. Garrantzitsua da kontroleko erantzuna sistemaren aldaketei egokitzea; osagai deribatibo haundi batek aldaketa handia dakar eta kontroladoreak horrekin bat erantzun dezake.

Hona hemen osagai deribatiboaren formula:

$$D_{out} = K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t) \tag{5.6}$$

non:

- D_{out}: kontroladore deribatiboaren irteera.
- K_d: irabazi deribatiboa.
- t: Denbora.
- $e(t)$: t unean prozesuaren errorea. $e(t) = SP - PV$.

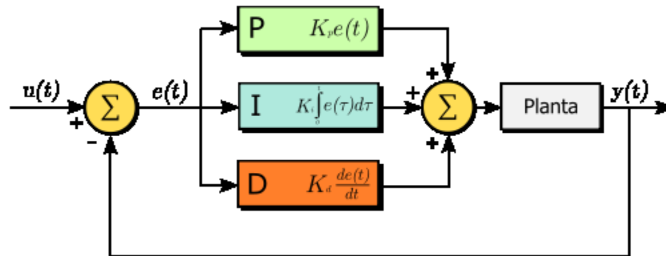
Hiru termino hauen irteerak (proportzionala, integrala eta deribatiboa) batu egiten dira PID kontroladorearen irteera kalkulatzeko. Demagun $u(t)$ dela kontroladoraren irteera, PID algoritmoaren azken forma ondorengoa litzateke:

$$\mathrm{u}(t) = \mathrm{MV}(t) = K_p e(t) + K_i \int_0^t e(\tau)\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t) \tag{5.7}$$

non:

- MV: manipulatutako aldagaia.

5.13 irudian ikusi dezakegu fisikoki nola antolatuko genukeen PID kontrol arkitektura.



Irudia 5.13: PID kontrol baten bloke-diagrama.

5.3.3 Kontroladorea idazten

Atal honetan gure kontroladorea idazteko eman ditugun urratsak ordenean aztertuko ditugu.

Lehenik eta behin, gure kontroladorea garatzeko pakete bat sortuko dugu, gure kasuan `mainbot_controller` izena duena. Pakete honen dependentziak `pr2_controller_interface`, `pr2_mechanism_model` eta `pluginlib` dira. `Pluginlib` paketeak gure kontroladorea *Controller Manager*-ean plugin bezala gehitzea ahalbidetzen du. Hona hemen emandako pausuak:

```
$ roscreate-pkg mainbot_controller
    pr2_controller_interface pr2_mechanism_model
    pluginlib
$ roscd mainbot_controller
$ rosmake
```

Honen amaieran dependentzia guztiak konpilatuta izango ditugu.

Irudia 5.14: Kontroladorearen garapenaren lehen fasea.

Ondoren, gure kodea idatziko dugu, Ackermann geometria kontutan hartuta komandoak jasotzen dituen arabera gure robota kontrolatzeko.

Aurrena, `include/mainbot_controller/mainbot_controller.h` fitxategia sortuko dugu gure `mainbot_controller` direktorioan. Bertan gure `MainbotControllerClass` klasea eta bere metodo eta aldagaiak definituko ditugu. Bere edukia zati garrantzitsuenak komentatuko ditugu:

Lehenik, hainbat dependentzia jarriko ditugu fitxategiaren goialdean. Dependentzia hauek PID kontrolerako, nabigaziorako, etab. beharrezkoak dira.

```
1 #include <pr2_controller_interface/controller.h>
2 #include <pr2_mechanism_model/joint.h>
3 #include <geometry_msgs/Twist.h>
4 #include <tf/transform_broadcaster.h>
5 #include <nav_msgs/Odometry.h>
6 #include <control_toolbox/pid.h>
7 #include <pr2_controllers_msgs/JointControllerState.h>
```

Orain, kontrolatu behar ditugun artikulazio bakoitzarentzat objektu bat definituko dugu, `pr2_mechanism_model::JointState` motakoa.

```
1   pr2_mechanism_model::JointState* joint_state_flw_dir_; // Aurreko
        ezkerro norabidea
2   ...
3   pr2_mechanism_model::JointState* joint_state_flw_; // Aurreko ezkerreko
        gurpila
4   ...
```

Geratzen diren definizio guztiak ikusteko **mainbot_controller.h** fitxategia begira dezakegu.

Mainbot_controller.cpp fitxategiaren inplementazioari dagokionez, 5.2.1 azpiatalean ikusi dugun bezala *init*, *starting*, *update* eta *stop* metodoak inplementatu behar ditugu. Gure kasuan, *commandCallback* izeneko metodo laguntzaile bat erabili dugu, oinarriak biratu beharreko angeluan mugak ezartzeko. Hona hemen inplementatutako metodoak:

- **init:** lehenik eta behin espero diren *joint* guztiak robotean daudela egiaztatuko dugu, aurrerago arazorik ez izateko. Ondoren, *joint* bakoitzaren PID kontrola eramango duten objektuak hasieratuko ditugu,

bat *joint* bakoitzarentzat. Jarraitzeko, hainbat aldagaiei balioak eman ondoren, **/cmd_vel** topic-era harpidetuko gara `geometry_msgs::Twist` motako mezuak entzuteko.

Mezu hauek entzundakoan `MainbotControllerClass::commandCallback` metodoari pasako zaizkio. Azkenik, **/joint_controller_states** topic-ean `pr2_controllers_msgs::JointControllerState` motako mezuak publikatuko ditugu, aurrerago eragingailuak *joint*-en bitartez kontrolatzeko.

- **starting:** metodo honetan kontroladorearen lehen zikloaren denbora gordeko dugu lehenik, aurrerago PID kontrolean erabiltzeko. Hortaz aparte, PID kontrola egingo duten objektu guztiak berhasieratuko ditugu.
- **update:** lehenago esan dugun bezala, kontroladorearen benetako kontrol-lana metodo honetan egingo dugu. Aurrena, gurpilek daramaten abiadura erreala metro/segundotan gordeko dugu (ROSek radian/segundotan ematen digu). Ondoren, Ackermann geometria kontutan hartuta *ICR*-a kalkulatuko dugu, hau da, gurpil guztiak komunean izango duten puntuaren erradioa. Jarraitzeko, barneko eta kanpoko aurreko gurpilek izan beharreko angelua kalkulatuko dugu, eta dagozkien moduan asignazioak egin. Orain, gurpilek eraman behar duten abiadura berria (jasotako komandoaren arabera) kalkulatu behar dugu, radian/-segundotan. Kalkulu hori egin ondoren, PID objektuen laguntzarekin komandatu beharreko esfortzuak asignatuko ditugu `JointState` objektuetan, eta azkenik, */joint_controller_states* topic-ean publikatuko ditugu.
- **commandCallback:** metodo honetan nagusiki komandatu beharreko angelua 0.7
- **stopping:** metodo honetan ez da beharrezkoa izan ezer inplementatzea, beraz, hutsik utzi dugu.

Behin kodea idatzi dugula, konpilatu egin behar dugu orain. Horretarako *CMakeLists.txt* fitxategian ondorengo lerroa gehitu dugu:

```
rosbuild_add_library(mainbot_controller_lib
         src/mainbot_controller.cpp
```

Ondoren, `make` exekutatuko dugu, horrela /lib direktorioan `mainbot_controller_lib.so` liburutegi fitxategi bat lortu dugularik.

Kodea liburutegi bezala konpilatu ondoren, bigarren fasera pasatuko gara (5.15 irudia).

Irudia 5.15: Kontroladorearen garapenaren bigarren fasea.

Kontroladorea `pr2_controller_manager`, `pluginlib` eta denbora errealeko prozesuek ikusgarri izan dezaten, kontroladorea duen paketeak klasea exportatu behar du. Horregatik, ondorengo komandoa jarri dugu kodearen fitxategiaren amaieran:

```
1 // Register controller to pluginlib
2 PLUGINLIB_DECLARE_CLASS(mainbot_controller,MainbotControllerPlugin,
3                          mainbot_controller_ns::MainbotControllerClass,
4                          pr2_controller_interface::Controller)
```

Ondoren, `make` komandoarekin kontroladorea konpilatu dugu. Orain, kontroladorearen paketearen dependentziak jarri behar ditugu `manifest` `-.xml` fitxategian, eta baita pluginaren deskribapen-fitxategia exportatu. Azpian doazen lerroak gehitu dira `manifest`
`-.xml` fitxategian:

```
1   <depend package="pr2_controller_interface"/>
2   <depend package="pr2_mechanism_model"/>
3   <depend package="pluginlib"/>
4   <depend package="roscpp"/>
5   <depend package="nav_msgs"/>
6   <depend package="control_toolbox"/>
7   <depend package="pr2_controllers_msgs"/>
8   <export>
9     <pr2_controller_interface plugin="\${prefix}/controller_plugins.xml"/
        >
10  </export>
```

Azkenik, aipatu dugun pluginaren deskribapen fitxategia sortu dugu.

```
1 <library path="lib/libmainbot_controller_lib">
2   <class name="mainbot_controller/MainbotControllerPlugin"
3          type="mainbot_controller_ns::MainbotControllerClass"
4          base_class_type="pr2_controller_interface::Controller" />
5 </library>
```

Prozesu hau jarraitu ondoren, gure kontroladorea exekutatzeko prest dago.

Kontroladorearen exekuzioa

Kontroladorea exekutatzeko bi aukera daude. Bat hasieraketa guztiak (*joint*-ak kontroladoreari lotu, kontroladorea kargatu eta hasieratu...) eskuz egitea

da. Lan guzti hau eskuz egitea oso nekeza denez, komenigarria da konfigurazioa fitxategi batean gordetzea hasieraketa guztiak eta, ondoren, *launch* fitxategi baten laguntzarekin exekutatu beharreko nodo guztiak exekutatzea.

Lehenik eta behin, aipatu dugun konfigurazioa fitxategian (`mainbot_-controller.yaml`) gure kontroladorearen mota, *joint*-en izenak, eta PID kontrolaren parametroak gordeko ditugu. Hemen behean ikusi dezakegu fitxategiaren edukia:

Irudia 5.16: Kontroladorearen garapenaren azken fasea.

```
1 mainbot_controller:
2      type: mainbot_controller/MainbotControllerPlugin
3      joint_name_flw_dir: joint_front_left_wheel_dir
4      joint_name_frw_dir: joint_front_right_wheel_dir
5      joint_name_blw_dir: joint_back_left_wheel_dir
6      joint_name_brw_dir: joint_back_right_wheel_dir
7      joint_name_blw: joint_back_left_wheel
8      joint_name_flw: joint_front_left_wheel
9      joint_name_brw: joint_back_right_wheel
10     joint_name_frw: joint_front_right_wheel
11     pid_parameters_dir:
12        p: 800.0
13        i: 0.0
14        d: 0.0
15        i_clamp: 0.0
16     pid_parameters_wheel:
17        p: 100.0
18        i: 0.0
19        d: 0.0
20        i_clamp: 0.0
```

Ondoren, robota hasieratzen duen *launch* fitxategian *Controller Manager*-a hasieratzea gehituko dugu amaieran, konfigurazio fitxategian zehaztu dugun konfigurazioarekin. Aprobetxatuz, robotaren hasieraketa guztia zehaztuko dugu jarraian.

Aurrena, URDF eredua parametroen zerbitzarian kargatuko dugu `robot-_description` aldagaia ondoren Gazebon kargatzeko. Segidan, **robot_-state_publisher** bat hasieratuko dugu, robotaren transformatuak publikatzeko (TF paketea erabiliz). Azkenik, kontroladorearen konfigurazio fitxategian parametroen zerbitzarian kargatuko dugu eta, jarraian, *Controller Manager*-a. Beraz, gure kodea horrela geratuko da:

```
1 <launch>
```

```
2
3    <!-- send mainbot.urdf to parameter server -->
4
5    <param name="robot_description"
6           command="$(find xacro)/xacro.py '$(find mainbot_description)/
                urdf/mainbot.urdf'" />
7
8    <!-- spawn robot in gazebo -->
9
10   <node name="spawn_object" pkg="gazebo" type="spawn_model" args="-urdf -
         param robot_description -y -4 -z 1.0 -model mainbot" respawn="false
         " output="screen" />
11
12
13   <!-- TF data (robot_state_publisher) -->
14  <node name="robot_st_pub" pkg="robot_state_publisher" type="
        state_publisher" />
15
16  <rosparam file="\$(find mainbot_controller)/mainbot_controller.yaml"
        command="load"/>
17  <node name="mainbot_controller_spawner" pkg="pr2_controller_manager"
        type="spawner" args="mainbot_controller" respawn="false" output="
        screen" />
18
19 </launch>
```

5.3.4 Gazebo Controller Manager plugina

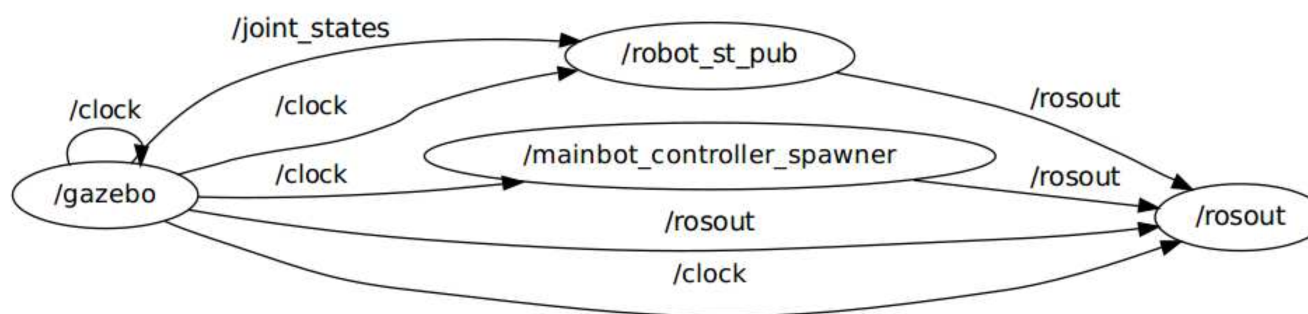
Azkenik, aipatu beharra dago gure lana Gazebon egiten ari garenez, `pr2_mechanism_model`-en ikusi dugun bezala *Controller Manager*-aren barnean dauden *joint*-etaz aparte Gazebon beste *joint* 'faltsu' batzuk ditugula.

Hauek nolabait elkartu behar dira, hau da, Gazebo eta Controller Managerraren arteko interfaze bat behar dugu hauek elkar ulertzeko gai izan daitezen.

Horretarako, **gazebo_ros_controller_manager** Gazeboko plugina erabili dugu.

5.4 Exekuzioa

Amaitzeko, 4. ataleko ingurune simulatua eta atal honetan ikusi dugun robotaren simulazioa abiatu ondoren izango ditugun nodoak eta hauen arteko erlazioak ikus ditzakegu 5.17 irudian. Bertan bi nodo berri azaltzen dira: */robot_st_publisher* eta */mainbot_controller_spawner*. Aurrenekoak robotaren transformatuak argitaratuko ditu */tf topic*-aren bitartez (irudi honetan ez da *topic* hori ikusten ez dagoela harpidedunik) eta bigarrengoa robotaren kontroladorea hasieratzeko nodo laguntzaile bat da, behin kontroladorea kargatu ondoren ezer egiten ez duena.



Irudia 5.17: Ingurune simulatuaren eta robotaren simulazioaren exekuzio garaiko nodoak.

6 Kapitulua

Nabigazioa

Gaien Aurkibidea

6.1 Sarrera **73**
6.2 Odometria **73**
6.3 Nabigazioaren mapa **75**
6.3.1 Irudi-fitxategia 75
6.3.2 YAML fitxategia 76
6.4 Move_base paketea **77**
6.4.1 Berreskuratze-portaerak (*Recovery behaviors*) . . . 77
6.5 Kostu-mapak **79**
6.5.1 Sarrera 79
6.5.2 Azalpen orokorra 80
6.5.3 Markaketa eta garbiketa 80
6.5.4 Espazio okupatua, librea eta ezezaguna 80
6.5.5 tf 80
6.5.6 *Puztea* 81
6.5.7 Mapa motak 81
6.5.8 Gure bi kostu-mapen arteko parametro komunak . 83
6.5.9 Kostu-mapa globala 84
6.5.10 Kostu-mapa lokala 84
6.6 Planifikatzaile globala **84**
6.7 Planifikatzaile lokala **85**
6.7.1 Gelaxka-sare mapa 86

6.7.2 Gure konfigurazioa 86
6.8 Nabigazio-sistemaren exekuzioa 87
6.8.1 Launch fitxategia 87
6.8.2 RViz tresna 88
6.8.3 Nabigazio komandoak bidaltzea 89
6.9 Exekuzioa 90

6.1 Sarrera

Gure robotaren nabigazioa inplementatzeko ROSeko *navigation* pila erabiliko dugu. Pila hau maila kontzeptualean nahiko sinplea da. Odometria eta sentsoreen informazioa hartzen du, ondoren oinarri mugikor bati abiadura aginduak bidaltzeko. Pila hau hurrengo hiru baldintzak betetzen dituen edozein robotak erabili dezake:

1. **ROS** exekutatzea robotak.
2. **`tf`** paketea erabiltzea.
3. ROSeko mezu mota egokiak erabiliz sentsoreen datuak argitaratzea.

Gainera, *navigation* pila robot konkretu batean ondo funtzionatzeko bere forma eta dinamikara egokitu behar da. Aipatu beharra dago pila hau ez dagoela pentsatuta Ackermann motako robotetarako, soilik robot diferentzial eta holonomoetarako. Aurrerago ikusiko dugu zein erabaki hartu diren arazo honi aurre egiteko.

6.2 Odometria

Navigation pilak ezartzen duen baldintzetako bat robotak odometria informazioa `tf` *topic*-ean publikatzea eta beste *topic* batean `nav_msgs/Odom-etry` motako mezuen bidez publikatzea da. Arestian aipatu dugunez, gure robotak GPS antena bat erabiltzen du lokalizazio-sistema bezala eta, ondorioz, ez du odometria kalkulurik egin behar. Beraz, atal honi odometria deitzea ROSeko *navigation* pilaren konbentzioak jarraitzeko erabakia jarraitzea da.

Errepaso txiki bat eginez, orain arte `/tf` *topic*-ean soilik robotaren beraren *frame*-en arteko transformatuak publikatzen ditugula gogoratuko dugu. *Navigation* pilaren beharrak asetzeko, `/map` eta `/odom` *frame*-ak robotaren *frame*-ekin erlazionatu behar ditugu, uneoro robota mapan kokatzeko gai izan gaitezen. Gogoratu behar dugu, baita ere, P3D pluginak robotaren kokapena `/base_ground_truth` *topic*-ean publikatzen duela. Hortaz, informazio hori erlazionatu behar dugu moduren batean.

Behar horretarako `mainbot_odometry` izeneko nodo bat idaztea erabaki dugu. Nodo honek egingo duena, funtsean, ondorengoa da, exekutatzen den ziklo bakoitzeko P3D pluginetik jasotako informazioa erabiliko dugu aipatutako informazioa toki egokian publikatzeko. Aurrena, `/odom` *frame*-tik `/base_footprint` *frame*-ra doan transformatua argitaratuko dugu `/tf` *topic*-ean.

Azkenik, `/odom` *topic*-ean `/map` *frame*-tik `/odom` *frame*-ra doan `nav_msgs::Odometry` motako mezua argitaratuko dugu.

Ikus dezagun kodea pausoz pauso azalduta:

```
1 #include <ros/ros.h>
2 #include <tf/transform_broadcaster.h>
3 #include <nav_msgs/Odometry.h>
4
5
6 nav_msgs::Odometry odom;
7
8 // /base_ground_truth topic-ean publikatzen den bakoitzean exekutatuko
      den Callback funtzioa
9 void p3dCallback( const nav_msgs::Odometry& p3d_odom){
10
11   odom = p3d_odom;
12
13 }
14
15 int main(int argc, char** argv){
16
17   ROS_INFO("mainbot_odometry node loaded!");
18   ros::init(argc, argv, "mainbot_odometry");
19   ros::NodeHandle n;
20   double frequency = 100.0; //Hz
21
22   // /base_ground_truth nodora harpidetuko gara
23   ros::Subscriber p3d_subscriber = n.subscribe("/base_ground_truth", 1,
         p3dCallback);
24
25   // /odom nodoan publikatzeko objektu bat sortuko dugu
26   ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("/odom",
         1000);
27
28   // /tf argitaratzaile bat sortuko dugu
29   tf::TransformBroadcaster odom_broadcaster;
30
31   ros::Time current_time, last_time;
32   current_time = ros::Time::now();
33   last_time = ros::Time::now();
34
35   ros::Rate r(frequency);
36   while(n.ok()){
37
38     ros::Time current_time;
39     ros::Time last_time;
40     current_time = ros::Time::now();
41     last_time = ros::Time::now();
42
43      //lehenik, transformatua tf bidez argitaratuko dugu
44     geometry_msgs::TransformStamped odom_trans;
45     odom_trans.header.stamp = current_time;
46     odom_trans.header.frame_id = "odom";
47     odom_trans.child_frame_id = "base_footprint";
48
49     odom_trans.transform.translation.x = 0.0;
50     odom_trans.transform.translation.y = 0.0;
51     odom_trans.transform.translation.z = 0.0;
52     odom_trans.transform.rotation.x = 0.0;
53     odom_trans.transform.rotation.y = 0.0;
54     odom_trans.transform.rotation.z = 0.0;
```

```
55      odom_trans.transform.rotation.w = 1.0;
56
57      // transformatua tf bidez bidali
58      odom_broadcaster.sendTransform(odom_trans);
59
60      // map frametik odom framera doan transformatua
61
62      odom_trans.header.frame_id= "map";
63      odom_trans.child_frame_id = "odom";
64
65      odom_trans.transform.translation.x = odom.pose.pose.position.x;
66      odom_trans.transform.translation.y = odom.pose.pose.position.y;
67      odom_trans.transform.translation.z = odom.pose.pose.position.z;
68      odom_trans.transform.rotation.x = odom.pose.pose.orientation.x;
69      odom_trans.transform.rotation.y = odom.pose.pose.orientation.y;
70      odom_trans.transform.rotation.z = odom.pose.pose.orientation.z;
71      odom_trans.transform.rotation.w = odom.pose.pose.orientation.w;
72
73      odom_broadcaster.sendTransform(odom_trans);
74
75      //ondoren, odometria mezua argitaratuko dugu ROSera
76      odom.header.stamp = current_time;
77      odom.header.frame_id = "odom";
78      odom.child_frame_id = "base_footprint";
79      odom_pub.publish(odom);
80
81      last_time = current_time;
82
83      r.sleep();
84      ros::spinOnce(); // deitzeko geratu diren callback funtzioak
            exekutatzeko
85    }
86
87  }
```

6.3 Nabigazioaren mapa

Navigation pilak mapak erabiltzeko aukera ematen du `map_server` eta honi lotutako tresnak erabiliz. Mapa bat definitzeko bi osagai behar dira. Batetik, maparen metadatuak deskribatzeko YAML fitxategi bat; bestetik, maparen okupazioa adieraziko duen irudi-fitxategi bat.

6.3.1 Irudi-fitxategia

Irudi-fitxategiak munduaren okupazio egoera zehazten du pixel bakoitzaren kolorearekin. Pixel txurienak eremu librea zehazten dute, beltzek okupatuta dagoela, eta tartekoak ezezagunak dira. Koloreak edo grisen eskalak onartzen dira, baina mapa gehienak grisak izan ohi dira. Ondoren, YAML fitxategian atalaseak defini ditzakegu hiru kategoria horiek desberdintzeko. Azken hau `map_server`-en barruan egiten da.

Atalase parametro batzuk izanda, pixel baten okupazio probabilitatea hurrengo ekuazioaren bidez kalkulatzen da:

$$OCC = \frac{255 - batazbesteko_kolorea}{255.0} \tag{6.1}$$

6.1 ekuazioan azaltzen den `batazbesteko_kolorea` kalkulatzeko irudiaren kolore guztien batazbestekoa egiten da. Okupazioa ROS mezuen bidez komunikatzen denean [0,100] motako osoko bat bezela adierazten da: 0 guztiz aske dagoenean, eta 100 guztiz okupatuta (-1 balioa erabat ezaguna denerako).

Gure mapa sortzeko GIMP tresnaz baliatu gara. Bertan robotak erabiliko duen errepidea txuriz marraztu dugu eta, beste guztia, beltzez. 6.1 irudian ikusi dezakegu gure mapa.

Irudia 6.1: Torresol zentralaren nabigaziorako mapa.

6.3.2 YAML fitxategia

Maparen YAML fitxategia hainbat parametro gordetzen dituen fitxategi sinple bat da. Bere eremuak ondorengoak dira:

- **image**: maparen irudi-fitxategiaren izena.
- **resolution**: maparen bereizmena, metro/pixel unitatetan.
- **origin**: maparen jatorria.
- **occupied_thresh**: okupatutako eremuen atalasea (hori baina okupazio probabilitatea handiagoa duten pixelak okupatutzat hartuko dira).
- **free_tresh**: eremu librearen atalasea (hori baino okupazio probabilitate txikiagoa duten pixelak libre daudela suposatuko da).
- **negate**: txuri/beltz edo libre/okupatuta dauden pixelen balioak alderantzizkatzeko.

Gure maparen YAML fitxategia definitzeko, jatorria aldatu behar izan dugu, Gazeboko mundu simulatuarekin bat etortzeko. Soilik 0 eta 1 balioak erabili dira irudi-fitxategia definitzeko, beraz atalaseak gutxi gorabeherakoak dira. Azkenik, erabili dugun maparen bereizmena 0.5 izan da.

```
1 image: torresol.pgm
2 resolution: 0.05
3 origin: [-163.3, -21.8, 0.0]
4 occupied_thresh: 0.8
5 free_thresh: 0.3
6 negate: 0
```

6.4 Move_base paketea

Nabigazio-sistemaren atalik garrantzitsuena `move_base` paketea da. Funtsean, mundu errealeko kokapen bat helburutzat emanda oinarri mugikor bat kokapen horretara eramaten saiatuko da. Nodo honek nabigazio ataza orokorra gauzatzeko planifikatzaile globala eta lokala erabiliko ditu. `nav_core::BaseGlobalPlanner` interfazea erabiltzen duen edozein planifikatzaile global erabili daiteke eta `nav_core::BaseLocalPlanner` interfazera egokitzen den edozein planifikatzaile lokal. Gainera, nodo honek bi kostu-mapa (costmap) erabiltzen ditu, bat planifikatzaile globalarentzat, eta beste bat planifikatzaile lokalarentzat. Hau dena aurrerago aztertuko dugu.

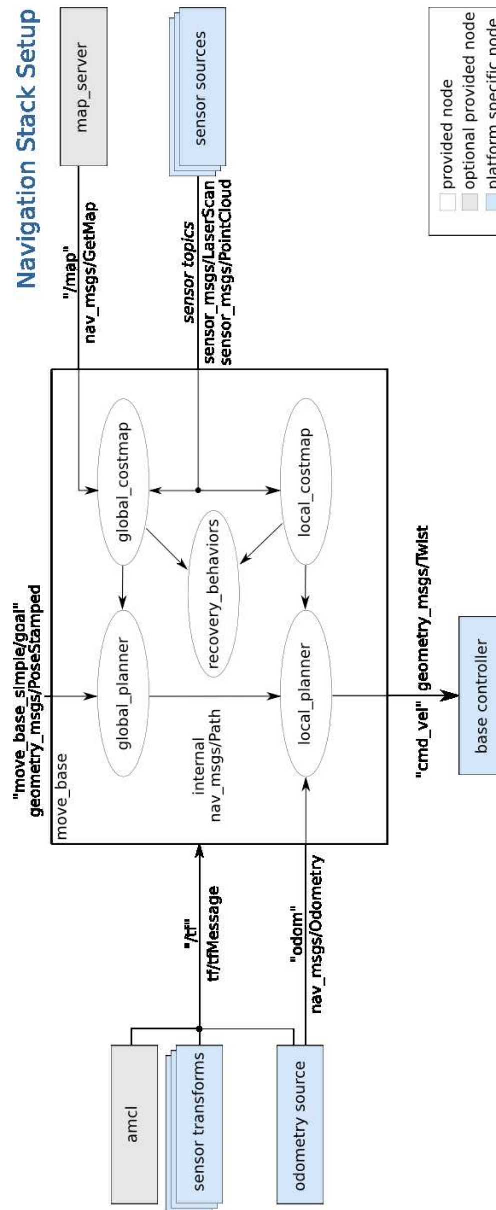
Aurrena, `move_base` paketeko `move_base` nodoaren konfigurazio orokorra ikusiko dugu eta, ondoren, gure kasuistikara nola egokitu dugun. 6.2 irudian ikusi dezakegu grafikoki nodoaren konfigurazioa.

Irudian ikusten den bezala hainbat elementu izan behar ditu ikusgai `move_base` nodoak: oinarriaren kontroladorea, sentsore-iturria (gure kasuan laserra izango dena), odometria-iturria `mainbot_odometry` nodoak emango diguna, robotaren transformatuak (`robot_state_publisher` eta `mainbot_odometry` nodoetatik lortuko duguna) eta, azkenik, hautazkoa den maparen zerbitzaria. Aipatzekoa da gure kasuan `amcl` nodoa ez dela beharrezkoa gure lokalizazioa zehatza izango delako GPS bat simulatzen ari garen heinean.

Nodoari posizio helburuak bidaltzeko `/move_base_simple/goal` *topic*-ean `geometry_msgs::PoseStamped` motako mezu bat argitaratuko dugu helburuaren posizio zehatza izango duena. Ondoren, `move_base` nodoak `/cmd_vel` *topic*-ean oinarri mugikorraren kontroladoreari abiadura-komandoak bidaliko dizkio.

6.4.1 Berreskuratze-portaerak (*Recovery behaviors*)

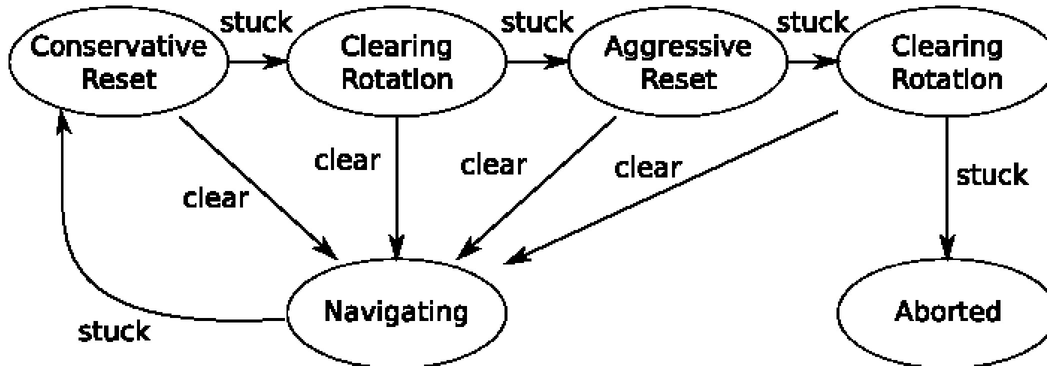
Egoki konfiguratutako robot bat gidatzen duenean, `move_base` nodoa helburu-posizio batera iristen saiatuko da, erabiltzaileak esandako tolerantzia bat aplikatuz. Oztopo dinamikorik ez dagoenean, `move_base` nodoak helburu horretara gidatuko du robota edo erabiltzaileari porrot seinale bat bi-



Irudia 6.2: *Navigation* pilaren egitura orokorra.

daliko dio. Robota punturen batean trabatuta geratzen bada berreskuratze-portaerak martxan jarriko ditu helbururako bideari ekiteko berriz.

6.3 irudian ikusten den bezala, erabiltzaileak ezarritako eskualde baten kanpoan dauden oztopoak robotaren mapatik ezabatuko ditu. Ondoren, posible bada, robotak bere burua biratuko du oztoporik gabeko bideren bat



Irudia 6.3: **move_base** nodoaren berreskuratze-portaerak.

aurkitzeko. Azken honek porrot egiten badu, robotak hasieratze bortitzago bat egingo du, eta robotak definitzen duen eremu errektangularraren kanpoan dauden oztopo guztiak ezabatuko ditu bere mapatik. Ondoren, beste bira bat emango du bere buruarekiko. Guzti honek porrot egiten badu, robotak bere helburua lortu ezintzat joko du eta erabiltzaileari helburua bertan behera utzi duela jakinaraziko dio.

Nabaria den bezala, gure robotak ezin du bere ardatzaren inguruan biratu Ackermann geometria erabiltzen duelako eta, hortaz, teorikoki ezin ditu berreskuratze-portaera hauek exekutatu. Horregatik, berreskuratze-portaerak desaktibatzeko erabakia hartu da. Horretarako `recovery_beha -vior_enabled` parametroari `false` balioa eman zaio.

6.5 Kostu-mapak

6.5.1 Sarrera

Kostu-mapak `costmap_2d` paketean daude inplementatuta. Sentsoreen informazioa hartuta 2 dimentsiotako okupazio mapa bat eraikitzen da, eta, ondoren, kostuak *puzten* dira okupazio informazioa eta erabiltzaileak zehaztutako *puzte* erradio bat kontutan hartuta. Guk nabigaziorako lehen azaldu dugun mapa erabiltzen dugunez, kostu-mapa eraikitzeko mapa hori erabiliko dugu.

6.5.2 Azalpen orokorra

costmap_2d paketeak egitura konfiguragarri bat eskaintzen du robotak non nabigatu behar duen (okupazio mapa baten bidez) jakiteko. Kostu-mapek sentsoreen informazio eta mapa estatikoa erabiltzen dute informazio-iturri bezala mundu errealeko oztopoen inguruko informazioa gordetzeko eta eguneratzeko. Hau dena costmap_2d::Costmap2DROS objektua erabiliz egiten da. Objektu honek bi dimentsiotako interfazea eskaintzen die erabiltzaileei, oztopoen inguruko galderak soilik zutabeka egin daitekeelarik. Hau da, jo dezagun mahai bat eta zapata bat digutula XY planoko posizio berdinean, baina Z posizio desberdinarekin. Objektu horretan posizio horri dagokion kostu-mapako gelaxkak kostu balio berdina izango luke. Hau espazio planoetan nabigatzea errazagoa izan dadin dago diseinatuta horrela.

6.5.3 Markaketa eta garbiketa

Kostu-mapa automatikoki harpidetzen da sentsoreek argitaratzen dituzten *topic*-etara, eta bere burua eguneratzen du hortik jasotako informazioaren arabera. Sentsore bakoitza markatzeko (kostu-mapan oztopo informazioa txertatzeko), garbitzeko (kostu-mapatik oztopo informazioa kentzeko), edo biak. Markaketa operazio bat array bat atzitzea da bertako gelaxka baten kostu balioa aldatzeko. Garbiketa operazio bat, aldiz, konplexuagoa da sentsoreen informazioa erabiliz azken honen jatorritik posizio jakin bateraino izpiak jarraitu behar direlako dagokion gelaxka garbi dagoen ala ez jakiteko.

6.5.4 Espazio okupatua, librea eta ezezaguna

Kostu-mapak update_frequency parametroan definitutako frekuentzian maparen eguneraketa zikloak beteko ditu. Ziklo bakoitzean, sentsoreen informazioa jasotzen da, ondoren markaketa eta garbiketa operazioak burutzeko eta, azkenik, egitura hau kostu-mapan proiektatzen da. Honen ondoren, oztopo bakoitzaren *puztea* burutzen da gelaxka bakoitzean. Operazio hau oso sinplea da; erabiltzaileak ezarritako *puzte* erradioa erabiliz okupatutako gelaxka bakoitzaren kostu-balioa kanpora hedatzen da. 6.5.6 azpiatalean ikusiko dugu prozesu hau zehatzago.

6.5.5 tf

Sentsore-iturrietako informazioa kostu-mapetan sartzeko, tf paketearen erabilera sakona egiten du costmap_2d::Costmap2DROS objektuak. Bere paketearen barruko global_frame (gure kasuan /map) eta robot_base_

`frame` (gure kasuan /base_footprint) parametroek zehazten dituzten *frame*-en arteko transformatu guztiak publikatzen direla suposatzen da, baita sentsore-iturrienak ere. `transform_tolerance` parametroan ezarriko da transformatuen elkarren arteko latentzia maximoa. Tf zuhaitza ez bada espero den frekuentzian eguneratzen, nabigazio pilak robota geldituko du.

6.5.6 *Puztea*

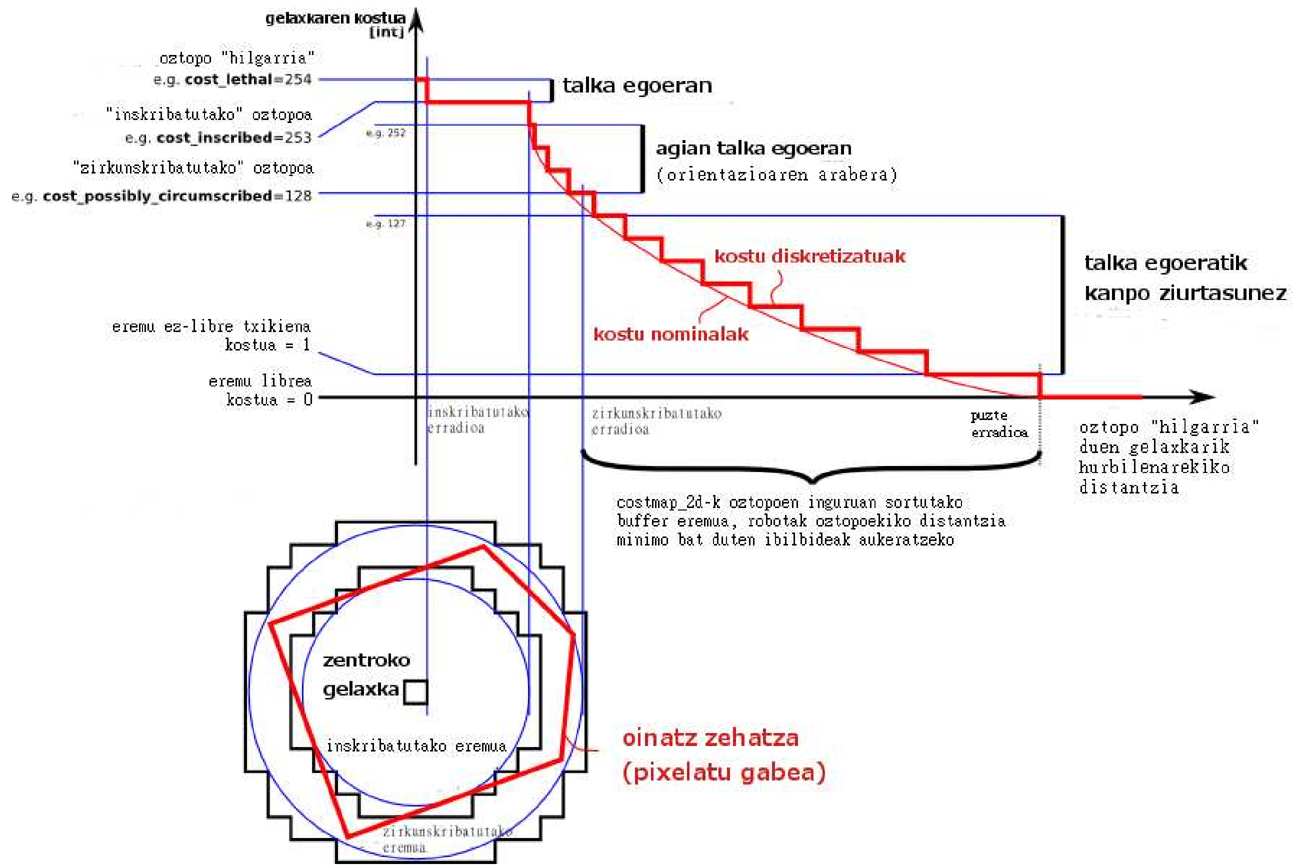
Puzte prozesuan okupatutako gelaxken balioak kanpora hedatzen dira. Ataza honetarako 5 kostu balio daude definituta kostu-mapan erabiltzeko:

- **Hilgarria** (lethal): balio hau duen gelaxkak benetazko oztopo bat adierazten du. Beraz, robotaren erdiko puntua gelaxka horretan egongo balitz kolisio egoeran egongo litzateke.
- **Inskribatua**: kostu balio hau duen gelaxka oztopo batetik inskribatutako erradioa baino distantzia txikiagora dago. Hortaz, robota seguraski kolisio egoeran egongo da bere erdiko puntua **inskribatutako** kostua edo balio handiagoa duen gelaxka batean baldin badago.
- **Zirkunskribatua agian**: kostu balio hau **inskribatua**ren antzekoa da, baina robotaren inskribapen erradioa erabili ordez, zirkunskribapen erradioa erabiltzen da. Hortaz, robotaren erdigunea balio hori duen gelaxka batean baldin badago robotaren noranzkoaren arabera oztopo bat jotzeko aukera izan dezake.
- **Espazio librea**: kostu hau zero bezala definitzen da, eta, horren arabera, robota modu askean ibili daiteke balio hau duten gelaxketan zehar.
- **Ezezaguna**: kostu hau duen gelaxkak ez du informaziorik ingurune errealean oztoporik dagoen ala ez jakiteko bere posizioan.
- Beste kostu guztiak **espazio librea** eta **zirkunskribatua agian** balioen artean egongo dira, **hilgarria** duen gelaxka batera duen distantzia eta erabiltzaileak ezarritako funtzio baten arabera.

6.4 irudian ikusi dezakegu balio hauen adierazpen grafikoa.

6.5.7 Mapa motak

`costmap_2d::Costmap2DROS` objektu bat hasieratzeko bi modu nagusi daude. Aurrenekoa, gure kostu-mapa globalean erabiliko duguna, mapa estatiko baten bidez hasieratzea izango da. Kasu honetan, kostu-mapa mapa



Irudia 6.4: Kostu-mapen gelaxken balioen aukeraketa.

estatikoak emandako zabalera, altuera eta oztopo informazioarekin hasieratuko da.

Bigarren moduan erabiltzaileak emango dio kostu-mapari zabalera eta altuera eta, ondoren, `rolling_window` parametroari *true* balioa emango dio. Azken parametro hau ezarrita, kostu-mapak robota izango du bere zentruan uneoro, eta oztopoei buruzko informazioa erabiltzaileak ezarritako eremu batera mugatuko da. Modu hau nabigazio lokalerako da egokia, robota bere inguru hurbilean dauden oztopoetaz bakarrik arduratu behar denerako. Hortaz, eremu txiki bat ezartzea izango da egokiena, robotak bere burutik urrun dauden oztopoei jaramonik ez egitea ez zaigulako axola maila honetan. Modu hau erabiliko dugu gure kostu-mapa lokalerako.

6.5.8 Gure bi kostu-mapen arteko parametro komunak

Lehenago azaldu dugun moduan, gure nabigazio-sistemak bi kostu-mapa izango ditu; bat planifikatzaile globalak erabiliko duena eta, bestea, planifikatzaile lokalak. Hala ere, bi kostu-mapek hainbat aldagai partekatzen dituztenez bien artean komunak diren parametroak komentatuko ditugu aurrena.

```
1 obstacle_range: 10.0
2 raytrace_range: 3.0
3 footprint: [[-1.2 ,-0.7], [1.2, -0.7], [1.5, 0.0], [1.2, 0.7], [-1.2,
      0.7]]
4 inflation_radius: 8.0
5
6 observation_sources: laser_scan_sensor
7 laser_scan_sensor: {sensor_frame: laser_link, data_type: LaserScan, topic
      : sick_laser_topic, marking: true, clearing: true,
      expected_update_rate: 10.0}
```

1 eta 2 lerroan dauden parametroek oztopoak mapetan ezartzeko atalaseak definitzen dituzte. `obstacle_range` parametroak oztopo bat robotaren zenbateko distantzia maximora egon behar duen ezartzen du oztopo hau kostu-mapan sartua izan dadin. `raytrace_range` parametroak, aldiz, oztopoa mapatik kentzeko egin behar diren operazioak gehiengo zein distantziara egingo dituen definitzen du.

3 eta 4 lerroei dagokienez, robotaren muga fisikoak definitzen ditugu. Lehenik, `footprint` parametroan gure robota mugatzen duen laukizuzena jarri dugu. Robotaren zentrua laukizuzenaren (0, 0) puntua izango da. Aipatu beharra dago, `footprint` parametroa erabili dugunez ez dela `robot_radius` parametroa definitu behar. Ondoren, *puzte*-erradioa definitu dugu. Parametro honekin definitzen duguna hurrengoa da, oztopoetatik erradio hori baina urrunago mantentzen saiatzea uneoro.

Azkenik, 6 eta 7 lerroetan gure laserrarekin lotzen ditugu bi kostu-mapak. Laser hau erabiliko dugu bi mapen markaketa eta garbiketa operazioak bu-

rutzeko eta 10 Hz-eko frekuentzian espero ditugula irakurketak ezarri dugu.

6.5.9 Kostu-mapa globala

Behin bi mapek izango dituzten parametro komunak aipatu ondoren, gure kostu-mapa globala azalduko dugu.

```
1 global_costmap:
2   global_frame: /map
3   robot_base_frame: /base_footprint
4   update_frequency: 5.0
5   static_map: true
```

Kostu-mapa global honetan azaltzen diren parametroei begiratzen badiegu, ikusi dezakegu `/map` *frame* globaletik `/robot_base_frame` *frame*-rako transformatua izango duela kontuan nabigazio-sistemak. Gainera, kostu-mapa globala segundoko 5 aldiz eguneratuko da eta, lehenago azaldu dugun moduan, mapa estatiko bat erabiliko dugunez `static_map` parametroari *true* balioa eman diogu.

6.5.10 Kostu-mapa lokala

Gure kostu-mapa lokala planifikatzaile lokalak erabiliko du oztopoen detekziorako, eta ez da gure mapan oinarrituko. Hona hemen bere konfigurazioa:

```
1  local_costmap:
2    global_frame: /map
3    robot_base_frame: /base_footprint
4    update_frequency: 5.0
5    publish_frequency: 2.0
6    static_map: false
7    rolling_window: true
8    width: 10.0
9    height: 10.0
10   resolution: 0.05
```

Kostu-mapa lokalak kostu-mapa globalaren *frame* berdinak izango ditu kontuan, eta segundoko bi aldiz publikatuko den arren, segundoko 5 aldiz eguneratuko da kostu-mapa hau. 10x10 metroko hedadura izango du robotaren erdigunea kontutan hartuta, eta bere bereizmena metro/pixel unitatetan 0.05ekoa izango da.

6.6 Planifikatzaile globala

Planifikatzaile globala (*global_planner*) nabigazio-sistemak erabiliko duen maila altuko planifikazioa sortzearen arduraduna izango da.

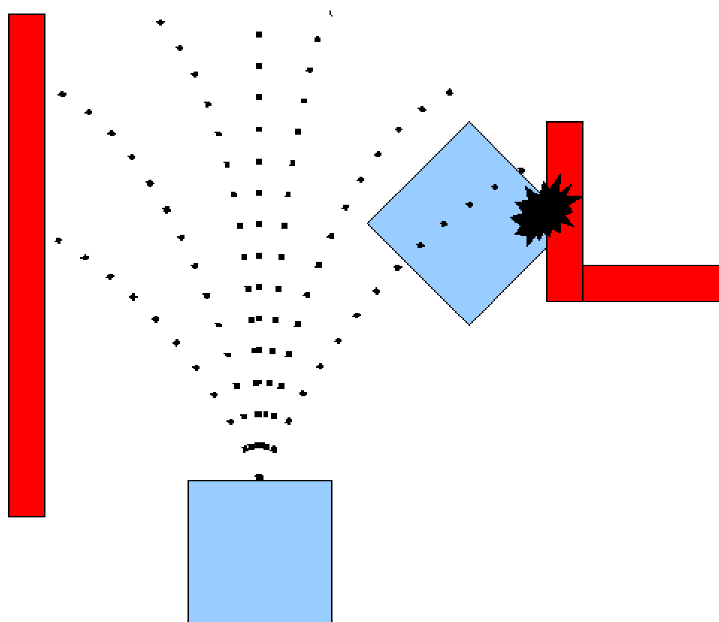
Gure planifikazio globalerako **navfn** paketea erabili dugu. Pakete honek oinarri mugikor batentzat planak sortzeko azkar interpolatutako nabigazio

funtzio bat ematen digu. Planifikatzaile honek robot zirkular bat espero du eta kostu-mapa bat erabiliz kostu minimoa duen hasiera puntu batetik bukaera puntu bateraino doan plan bat aurkituko du. Nabigazio funtzioak Dijkstra algoritmoa erabiltzen du.

6.7 Planifikatzaile lokala

Planifikatzaile lokala oinarri mugikorrari bidaliko zaizkion abiadura eta noranzko komandoak sortzearen arduraduna izango da, eta planifikazio lokal bat jarraituko dute komando hauek.

Planifikazio lokalerako `base_local_planner` paketea erabili dugu. Pakete honek oinarri mugikor bat plano batean gidatzeko kontroladore bat ematen du. Kontroladore honek planifikatzailea robotarekin lotzeko balio du. Mapa bat erabiliz, hasiera puntu batetik helburu puntu batera iristeko ibilbide zinematiko bat sortzen du planifikatzaile honek. Planifikatzaile honek gelaxka-sare bat sortzen du robotaren inguruan balio funtzio baten arabera. Balio-funtzio honek gelaxka-sarean zehar ibiltzearen kostua kodetzen du. Kontroladorearen lana izango da balio-funtzio hau erabiltzea robotari bidali behar zaizkion dx, dy eta $d\theta$ balioak kalkulatzeko.



Irudia 6.5: Planifikatzaile lokalaren ibilbide posibleak.

Algoritmo honen oinarrizko ideia ondorengoa da:

1. Robotaren kontrol espazioa (dx, dy eta $d\theta$) gelaxka diskretutan lagintzea.
2. Lagindutako abiadura bakoitzarentzat, lagindutako abiadura hori robotari denbora periodo labur batean aplikatzen zaiola simulatu.
3. Simulatutako ibilbide bakoitza ebaluatu hurrengo ezaugarriak kontuan izango dituen metrika bat erabiliz: oztopoekiko hurbiltasuna, helburuarekiko hurbiltasuna, ibilbide globalarekiko hurbiltasuna eta abiadura. Oztopoen kontra jotzen duten ibilbideak baztertu.
4. Ebaluazio handiena jaso duen ibilbidea hartu eta hari lotutakoo abiadura-komandoak oinarri mugikorrari bidali.
5. Berriz hasierara joan.

6.7.1 Gelaxka-sare mapa

Ibilbideak modu eraginkorrean ebaluatzeko gelaxka-sare mapa bat erabiltzen da. Kontrol-ziklo bakoitzerako, robotaren inguruan gelaxka-sare bat sortzen da (kostu-mapa lokalaren tamainakoa) eta ibilbide globala eremu honetara mugatzen da. Honen arabera, gelaxkaren batek ibilbide globalera 0 distantzia izango du, eta gelaxkaren batek helburu puntura 0 distantzia. Ondoren, zabaltze-algoritmo batek beste gelaxken balioak kalkulatuko ditu hurbilen dagoen 0 balio duen puntutik **Manhattan distantzia**[1] erabiliz.

Gelaxka-sarea ibilbideen ebaluaziorako erabiltzen da ondoren.

Bestetik, ibilbide globalaren helburua gelaxka-sare lokaletik kanpo geratzen bada ere, gelaxka-sare barruan dagoen ibilbideko azken puntuaren hurrengo puntua *helburu lokal* bezala konsideratzen da.

6.7.2 Gure konfigurazioa

Mainbot robotaren nabigaziorako planifikatzaile lokalaren konfigurazioa ez da konfigurazio estandarraren oso desberdina. Konfigurazio parametroei begirada bat botako diegu:

```
1 TrajectoryPlannerROS:
2   max_vel_x: 1.0
3   min_vel_x: 0.2
4
5   acc_lim_th: 10.0
6   acc_lim_x: 2.5
7   acc_lim_y: 2.5
8
```

[1] Ikus http://en.wikipedia.org/wiki/Taxicab_geometry webgunea.

```
 9  holonomic_robot: false
10
11  sim_time: 4.0
12  sim_granularity: 0.001
```

1 eta 2 lerroetan robotak hartu dezakeen abiadura maximoa eta minimoa zehaztu ditugu. Abiadura minimo bat ezarri diogu, Ackermann geometrian gabiltzanez eta hori nabigazio pilak kontutan hartzen ez duenez, abiadura angeluarra soilik izango lukeen abiadura komandorik bidali ez diezaion kontroladoreari. Gainera azelerazio maximoak ezarri ditugu aurrera, atzera eta norabidearentzako. Ondoren, `holonomic_robot` parametroari *false* balioa eman diogu. Parametro hau defektuzko balioan utzi izan bagenu (*true*), planifikatzaile lokalak alborakako mugimenduak (ingelesez *strafing*) bidaltzeko aukera izango luke. Azkenik, planifikatzaile lokalak aurreraka 4 segundoko simulazioa egiteko eskatu diogu eta ibilbideko puntuen artean 0.001 metroko distantzia egoteko.

6.8 Nabigazio-sistemaren exekuzioa

Behin nabigazio-sistemaren muina ikusita, bera nola exekutatu eta erabili azalduko dugu atal honetan. Bi zati izango ditugu hemen; bat nabigazio-sistema martxan jartzea, eta, bestea, nabigazio-sistemarekin lan egitea helburuak bidaltzeko eta bere informazioa ikusteko.

6.8.1 Launch fitxategia

Nabigazio-sistema martxan jartzeko, ROSeko `roslaunch` tresna baliagarria erabiliko dugu. Horrela, nodoak eta parametroak automatikoki hasieratu ahal izango ditugu. Nabigazio-sistema exekutatzeko hurrengo **.launch** fitxategia sortu dugu:

```
 1  <launch>
 2
 3  <!-- Run the map server -->
 4  <node name="map_server" pkg="map_server" type="map_server" args="$(find
         mainbot_2dnav)/maps/torresol.yaml"/>
 5
 6  <!-- Run move_base -->
 7  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
         output="screen">
 8
 9  <!-- Disable Recovery Behaviors -->
10  <param name="recovery_behavior_enabled" value="false"/>
11
12  <!-- Load Costmaps & Local planner params -->
13  <rosparam file="$(find mainbot_2dnav)/costmap_common_params.yaml"
        command="load" ns="global_costmap" />
14  <rosparam file="$(find mainbot_2dnav)/costmap_common_params.yaml"
        command="load" ns="local_costmap" />
```

```
15   <rosparam file="$(find mainbot_2dnav)/local_costmap_params.yaml"
         command="load" />
16   <rosparam file="$(find mainbot_2dnav)/global_costmap_params.yaml"
         command="load" />
17   <rosparam file="$(find mainbot_2dnav)/base_local_planner_params.yaml"
         command="load" />
18
19   </node>
20   </launch>
```

Ikus dezakegun bezala, lehenik mapen zerbitzaria hasieratzen dugu maparen YAML fitxategiarekin aurrerago `move_base` nodoak erabil dezan. Ondoren, `move_base` nodoa hasieratzen dugu. Hasieraketa honen barruan, berreskuratze-portaerak desgaituko ditugu aurrena eta, ondoren, kostu-mapen eta planifikatzaile lokalaren parametroak kargatuko ditugu.

6.8.2 RViz tresna

RViz tresna hainbat egoeratan baliagarria den 3D bistaratze-tresna bat da. Exekuzioko elementu anitzak elkarren artean nola portatzen diren jakiteko oso egokia da. Gure kasuan robota eta bere nabigazioan parte hartzen duten elementuak bistaratuko ditugu. Hauek dira bistaratuko ditugun elementuak:

- **Robotaren eredua**: parametroen zerbitzariko *robot_description* parametroa grafikoki bistaratuko da %30eko transparentziarekin.
- **Odometria**: `/odom` *topic*-ean publikatzen den robotaren odometria grafikoki gezi bat bezala bistaratuko da.
- **Planifikatzaile globalaren ibilbidea**: planifikatzaile globalaren ibilbidea irakurriko da `/move_base/NavfnROS/plan` *topic*-etik.
- **Planifikatzaile lokalaren ibilbidea**: planifikatzaile lokalaren ibilbidea irakurriko da `/move_base/TrajectoryPlannerROS/local_plan` *topic*-etik.
- **Helburu posizioa**: `move_base_simple/goal` *topic*-etik une honetako helburu-posizioa irakurriko da gezi bezala adierazteko.
- **Tf zuhaitza**: gure robotaren *link*-en arteko erlazioa adierazten duen tf zuhaitza grafikoki ikusi dezakegu. *Frame* bakoitza X (gorria), Y (berdea) eta Z (urdina) bezala adieraziko da.
- **Laser irakurketak**: gure laserraren izpi bakoitzaren balioa irakurriko da `sick_laser_topic` *topic*-etik. Irakurketa bakoitza koadro baten bidez adieraziko du, eta irakurketa berriak iristen diren heinean, aurrekoak ezabatuko dira automatikoki.

- **Robotaren oinatza**: robota zirkunskribatzen duen oinatza grafikoki ikusi dezakegu `/move_base/local_costmap/robot_footprint` *topic*-etik irakurrita.

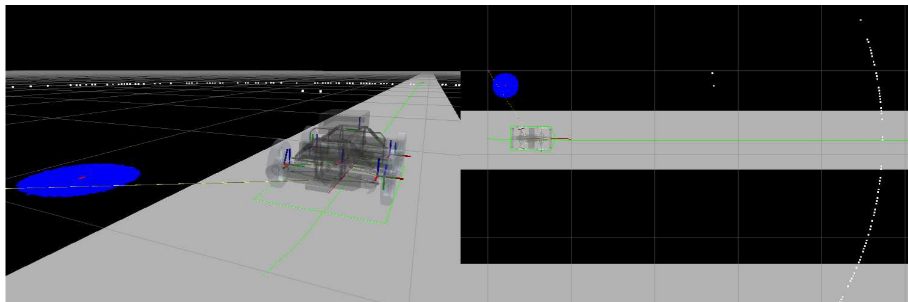
- **Nabigazioko mapa**: planifikatzaile globalak erabiltzen duen mapa `/map` *topic*-etik irakurriko dugu.

- **Oztopoak**: planifikatzaile lokalak aurkitzen dituen oztopoak kolore gorriz ikusiko ditugu, `move_base/local_costmap/obstacles` *topic*-a irakurrita.

- ***Puztutako* oztopoak**: oztopoen inguruan *puzten* den eremua kolore urdinez ikusiko dugu, `/move_base/local_costmap/inflated_obstacles` *topic*-a irakurrita.

- **Eremu ezezaguna**: planifikatzaile lokalak ezezagun bezala kontsideratzen duen eremua kolore horiz ikusiko dugu, `/move_base/local_costmap/unknown_space` *topic*-etik.

Azaldutakoa exekuzioan ikusteko aukera dugu 6.6 irudian.



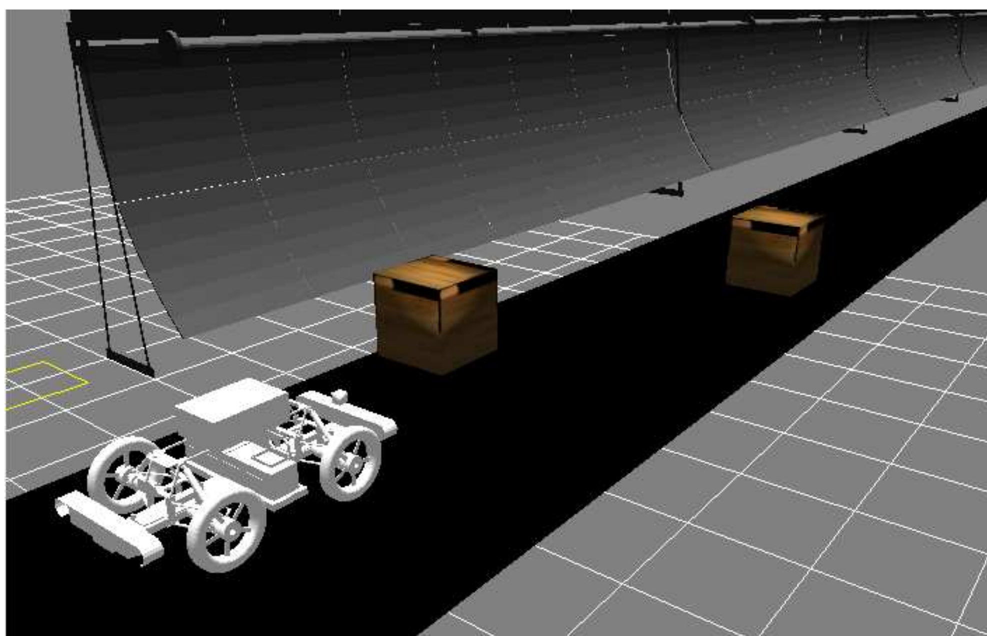
Irudia 6.6: Nabigazio-sistemaren bistaratzea RViz tresnan.

6.8.3 Nabigazio komandoak bidaltzea

Azken guzti hau ikusi ondoren, nabigazio-sistemari komandoak nola bidaltzen zaizkion ikasiko dugu. Bi modu daude horretarako, bat `actionlib` paketea erabiliz posizio helburuak bidaliko dituen nodo bat idaztea da, horretarako helmugara iritsitakoan abisatuko digun zerbitzu bat erabiliz. Beste bat, guk erabili duguna, `/move_base_simple/goal` *topic*-ean helburu posizioa publikatzea da. Gainera, *topic* horretan publikatzeko RViz tresnak

laguntza grafikoa eskaintzen digu, puntu geometriko hortan klik eginez agindua bidaltzea ahalbidetzen digulako.

Hortaz, robotaren simulazioa eta nabigazio-sistema zuzen dabiltzala egiaztatzeko proba txiki bat egingo dugu. Proba honetan, oztopoen detekzioa ondo egiten dela bermatzeko Gazebok eskaintzen dituen bi kutxa jarri ditugu robotaren ibilbidean. Kutxa horiek nahikoa bide uzten dute libre robota hauek maniobra txiki bat eginez sahiesteko aukera izan dezan. 6.7 irudian ikusi dezakegu hasierako konfigurazioa Gazebo simuladorean.

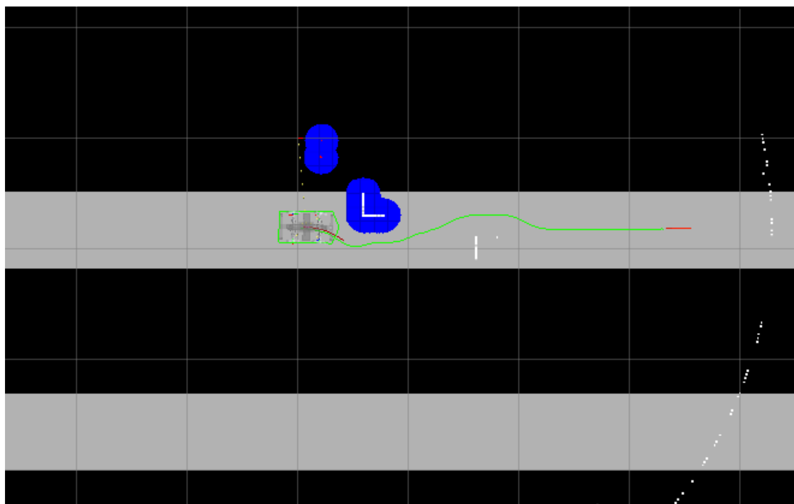


Irudia 6.7: Sistema robotikoaren exekuzio proba Gazebon ikusita.

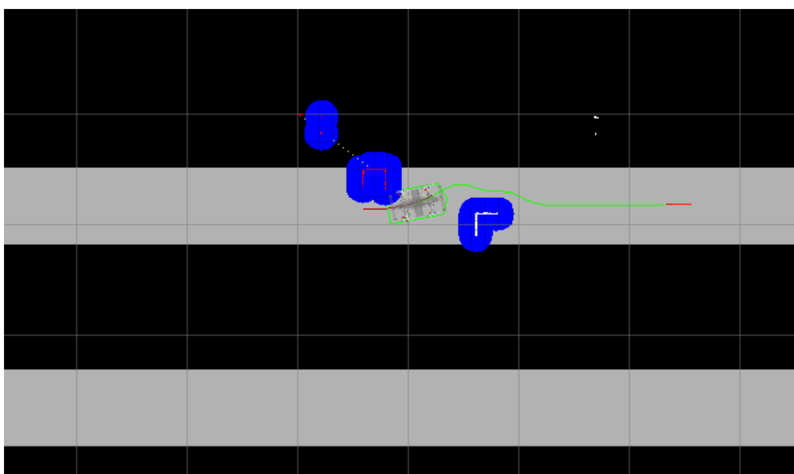
Nabigazio helburua ezartzeko, RViz-ek eskaintzen duen **2D Nav Goal** botoian sakatuko dugu eta, ondoren, gure helburuan gezi bat jarriko dugu norabidea eta noranzkoa ezartzeko. 6.8(a) , 6.8(b) eta 6.8(c) irudietan ikusi dezakegunez, oztopoak sahiestuko dituen plan global bat egiten du exekuzioa martxan jarri aurretik eta hori modu zuzenean exekutatzen du oztopoak ekidinez.

6.9 Exekuzioa

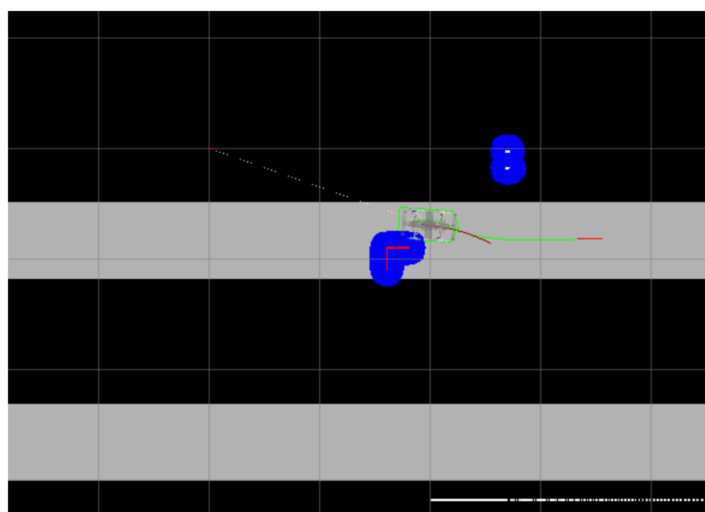
Amaitzeko, sistema guztia abiatu ondoren exekuzio garaian izango dugun grafoa ikus dezakegu 6.9 irudian. Bertan ikus dezakegu simulazioa eta nabigazio-sistemaren arteko hartuemana. Nabigazio-sistemak hiru nodo berri sortu



(a) Robotak bi oztopoak kontutan izango dituen bidea planifikatzen du.



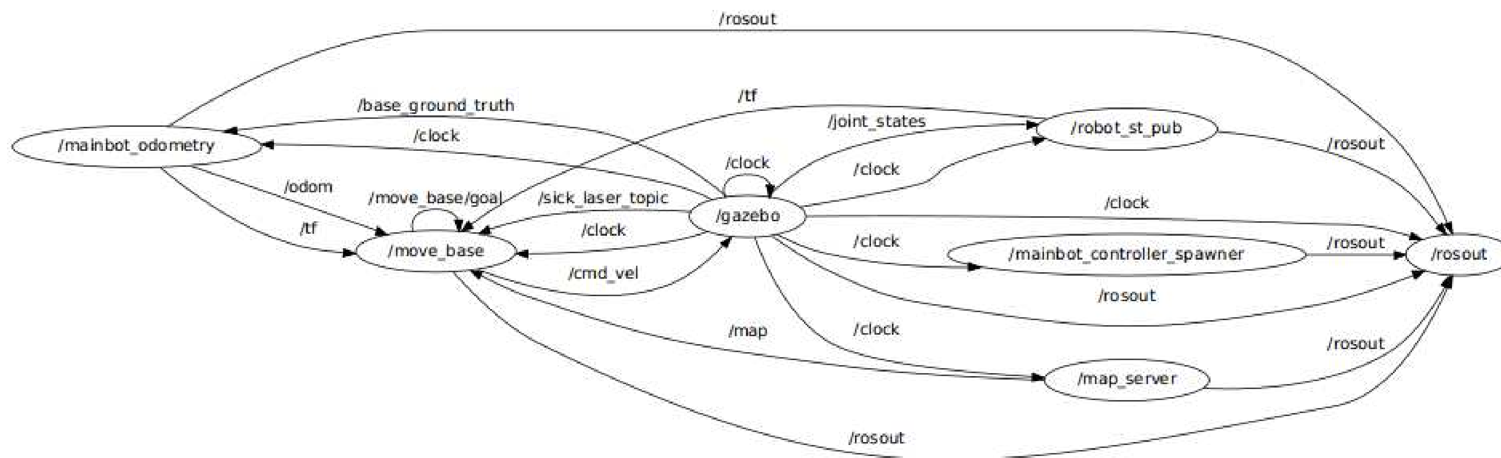
(b) Robota lehen eta bigarren oztopoaren erdian.



(c) Robotak hasieran aurrean zituen bi oztopoak gainditu ditu talkarik gabe.

Irudia 6.8: Sistema robotikoaren exekuzio proba RViz-en.

ditu: */mainbot_odometry*, */move_base* eta */map_server* izenekoak. Grafo honen bitartez errazago uler dezakegu sistema guztiak nola egiten duen elkar, exekuzio unean soilik gertatzen baita hau.



Irudia 6.9: Sistema osoaren exekuzio garaiko nodoak.

7 Kapitulua

Ondorioak eta etorkizunerako lana

Gaien Aurkibidea

7.1 Ondorioak **96**

7.1.1 Ondorio pertsonalak 97

7.2 Etorkizuneko lana **97**

Azken kapitulu honetan proiektuaren inguruko ondrioak azalduko dira, hauen artean: ondorio orokorrak eta ondorio pertsonalak. Honetaz gain, inplementatu gabeko eta proiektuaren etorkizunerako interesgarriak izan daitezkeen hainbat zeregin azalduko dira, gai guztiak jorratzeko denborarik ez baita izan.

7.1 Ondorioak

Proiektua amaituta, hasieran ezarritako helburuak bete direla esan diteke. Garatu diren ROSeko paketeei esker Mainbot robotaren simulazioa eta nabigazioa ikusi ditzakegu martxan ROS/Gazebon. Ondoren, proiektutik ateratako ondorioak aztertuko dira:

1. ROS *framework*-a erabiltzeak hainbat abantaila ematen dituen arren, ikasketa kurba handia du bere konplexutasuna dela eta. Hainbat kontzeptu barneratu behar dira ROS erabiltzen jakiteko, eta denbora handia eskatzera hel daiteke. Hala ere, kontzeptuak guztiz barneratuta izan gabe ere kodea garatzea ez dela ezinezkoa esan daiteke.

2. Robot baten kontroladorea idaztea oso konplexua izatera hel daiteke, eta garbi izan beharko dugu mugimendu-sistema definitzen duten ekuazioak zein diren. Kasu honetan, ROSen garatutako Ackermann robot gutxi daudenez laguntza gutxi aurkitu da, eta horregatik nahi baina denbora gehiago jardun da ataza honetan. Hala ere, esan dezakegu kontroladore txukuna lortu dela eta bere portaera egokia dela.

3. Robotaren nabigazioari dagokionez, ROSeko *navigation* paketea robot holonomoetarako pentsatuta dagoela kontutan hartuta, orokorrean ondo portatzen dela ikusi da. Hala ere, planifikatzaile globalak biraketa zabalak baditu robota aurrera joan ezinik geratzen da. Dena den, robota guk definitutako ingurunean mugituko dela kontutan hartzen badugu, planifikatzaileak ez ditu biraketa bortitzak izango, eta, biraketa batzuetan abiadura motela erakutsi dezakeen arren, bere helburu-puntura iritsi da erabiltzaileak egindako probetan.

4. Aipatu beharra dago, Mainbot robotaren ezaugarrietatik abiatutako simulazioa egin dugun arren, ikasleak ez duela simulazioaren portaera robot errealarekin alderatzeko aukera izan, eta, ondorioz, ezin dugu esan benetan errealitatearen antzeko portaera lortu dugunik.

7.1.1 Ondorio pertsonalak

Proiektu hau garatzeko ikasleak jardunaldi osoko arduraldia izan du IK4-Tekniker zentro teknologikoaren barnean, karrerako ikasgai guztiak amaituak zituelako alegia. Proiektu hau garatzeko ideia **Sistemen kontrola** irakasgaia egin ondoren sortu zen, robotikaren munduak ikaslean interesa sortu zuelako. Proiektu hau garatu ondoren hainbat ondorio pertsonal atera dira:

1. Enpresa munduan talde-lanak berebiziko garrantzia du. Norberak ezin du lan bat bere kaxa guztiz eraman aurrera, eta, horregatik, beste lankideekin zalantzak kontsultatzen jakitea ezinbestekoa da. Horregatik, gaitasun pertsonalak landu behar dira lana egoki egiteko.

2. Ikasleak lehenago praktikak egin zituen arren (IXA taldean, fakultatean alegia) kasu honetan enpresa mundura jauzia egiteak aldaketa handia suposatu dio. Etorkizunean baliagarria izango zaion lan-esperientzia izan du eta enpresen dinamikak eta jarduteko moduak gertutik ikusteko aukera izan du.

3. Hau izan da ikasleak garatu duen lehenengo proiektu handia, eta aurrera begira lan-metodologia bat ezartzeko lagundu dio.

4. Testuak LaTeXbidez sortzeko konpetentzia lortu du, eta horri esker itxura zientifikoa duten testuak sortzen ikasi du.

7.2 Etorkizuneko lana

Karrera bukaerako proiektua amaitutzat eman badugu ere, honen inguruan hainbat lan egin daitezke, etorkizunean egin ahal direnak, sistemaren portera fintzeko edo baliagarritasuna hobetzeko asmoz. Ondoren, hauen zerrenda bat ikusiko dugu:

1. Mainbot robotaren simulazioa robot errealaren portaerarekin alderatu dezakegu. Hau da, sistemaren portaera erakusten duten test batzuk definituz bi sistemek izandako emaitzak alderatu ditzakegu simulazioaren zehaztasuna ikusteko. Adibidez, robotari eta simulazioari abiadura komandoak bidali diezaizkiekegu eta bataren eta bestearen amaiera puntuak alderatu. Gainera, simulatutako laserra eta benetazko laserra alderatu ditzakegu, ingurune berdinean biek sortzen duten irteerek berdinak edo oso antzekoak izan beharko luketelarik.

2. Mainbot robot errealak dituen ultrasoinu-sentsoreak eta talka-sentsoreak gehi ditzakegu Mainbot robotak ingurunetik jasotzen duen informazioaren berdina izateko.

3. Nabigazio-sistemarako ROSeko *navigation* paketea arrakastaz erabili dugun arren, ez ditu Ackermann sistema batek dituen murrizketak kontuan hartzen. Gure proiektuak zehazten duen ingurunean honek hainbesteko garrantziarik ez duen arren, gure mugimendu-sistemaren murrizketak kontuan izango dituen planifikatzaile global bat idatzi dezakegu. Horretarako, ROSeko *Search Based Planner* (SBPL) liburutegia erabili dezakegu. SBPL planifikatzailea erabiltzeko robotaren mugimendu primitiboak (*motion primitives*) definitu beharko ditugu, eta, ondoren, planifikatzaileak mugimendu hauek robotak egin dezakeen mugimendu-unitate bezala erabilita hauez osatutako bide bat osatuko du. Planifikatzaile global hau izango bagenu, robota edozein motako ingurunean nabigatzeko gai izango litzateke.

4. Gure kasuan, robotaren nabigazio-sistemari helburu-puntu bat bidaltzeko RViz tresna erabiltzen dugu. Zehazki, robotari puntu batera joatea esateko, RViz-eko ingurunean puntu horretan klik egingo dugu eta robota horra joaten saiatuko da. Hala ere, posible da robotari helburuak kodea erabiliz bidaltzea, RViz-ek barnean egiten duen bezala. Horretarako, nodo berri bat sor genezake automatikoki helburuak bidaltzeko. Gainera, hainbat puntu aurredefiniturik izan ditzakegu (fitxategi batean, adibidez) eta nodo horren bitartez robotak puntuak hurrenez hurren bisitatzea lortu dezakegu.

Bibliografia

Artikuluak

[1] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Ng, *ROS: an open-source Robot Operating System*. Willow Garage.

Liburuak

[2] Maja J Matarić, *The Robotics Primer (2007)*. MIT Press

[3] Kastern Berns, Ewald von Puttkamer, *Autonomous Land Vehicles: Steps towards Service Robots*. Springer.

Interneteko erreferentziak

[4] ROS, *http://wiki.ros.org/ROS*.

[5] tf, *http://wiki.ros.org/tf*.

[6] Simulagailuak, *http://en.wikipedia.org/wikiRobotics_simulator*.

[7] Gazebo, *http://gazebo.willowgarage.com*.

[8] PID Controller, *http://en.wikipedia.org/wiki/PID_controller*.

[9] Ackermann mugimendu-sistema, *http://www.idsc.ethz.ch/Courses/vehicle_dynamics_and_design/11_0_0_Steering_Theroy.pdf*.

[10] LMS221 laserra, *https://www.mysick.com/saqqara/im0012759.pdf*

[11] Navigation, *http://wiki.ros.org/navigation*.