eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Informatika Ingeniaritzako Gradua

Konputagailuen Ingeniaritza

Gradu Amaierako Proiektua

Mover4 beso robotikoaren traiektoriak 3D ikusmen sentsorearen elikaduraz

---

Egilea

Daniel Camacho Piris

Zuzendariak

Elena Lazkano eta Txelo Ruiz

informatika fakultatea
facultad de informática

2017ko iraila

Laburpena

Honako proiektuan Robotika eta Sistema Autonomoen Ikerketa Taldeko (RSAIT) Mover4 beso robotikoa eta Microsoft-eko Kinect sentsorea erabili dira piezen manipulazioa gauzatzeko ROS, Moveit! eta rViz inguruneak erabiliz. CommonPlace Robotics enpresako Mover4 beso robotikoak Kinect sentsorea darabil manipulatu beharreko piezak eta talka-objektuak hautemateko. Informazio honetaz baliatuz beharrezko objektua manipulatu eta bidean aurkitzen dituen obstakuluak ekiditen ditu. Sistema hau Donostiako Informatika Fakultatean RSAIT ikerketa taldeak duen laborategian instalatu da

Gaien aurkibidea

Irudien aurkibidea

Taulen aurkibidea

Adibideak

1. KAPITULUA

Aurkezpena

1.1 Sarrera

Beso robotikoak faktorietan lan errepikakorrak arintzeko edota pertsonentzat astunegiak diren objektuak manipulatzeko erabiltzen diren gailuak dira. Robot hauek, noski, beraien kabuz ingurune batean eragin behar dute, gizakien inongo interbentziorik gabe. Horretarako, beraien inguruaz jabe izan behar dute, ingurua ulertu behar dute. Hori dela eta, askotan, ikusmen artifiziala gehitzen zaie honako sistemei hainbat sentsore desberdinen bidez, esaterako, 3D kamerak, Microsoft etxeko Kinect-a bezalakoak.

Aipatutako sistema hauek erreferentzi moduan harturik, proiektu honetan piezak manipulatzeko eta ingurune dinamikoan aritzeko gai den beso robotiko baten kontrola garatuko da. Sistema honek, piezak posizio batzuetatik beste batetara mugituko ditu, bere inguruan topa ditzazken oztopoak ekidinez.

Bete beharreko helburu guztiak asetzeko, CommonPlace Robotics etxeko Mover4 beso robotikoa, Kinect sentsorea, Moveit! mugimendu planifikaziorako softwarea, rViz bistaratzailea eta ROS (Robot Operating System/Robot Open Source) framework-a erabiliko dira.

1.2 Proiektuaren deskribapen orokorra

Lehen esan bezala, Mover4 beso robotikoak piezak manipulatuko ditu ROS framework-a, Moveit! softwarea eta rViz bistaratzailea erabiliz. Hori dela eta, proiektua hiru zatitan banatuko da: lehenengo fasean, ingurunea diseinatu eta instalazioa rViz simulatzailean errepresentatuko da, lan-ingurua zehaztuko da. Bigarrengoan, Moveit! softwarea erabiliz, Kinect sentsorearen laguntzaz, ingurunearen konszientzia garatuko dugu eta beso robotikoari oztopoak ekiditeko gaitasuna emango diogu. Azkeneko fasean, aldiz, piezen identifikazioari ekingo diogu, hau da, sentsore optikoaren bitartez piezaren posizioa hautemango da, geroago, Moveit! inguruneari esker beso robotikoak posizioen arteko mugimendua kalkulatu eta exekutatu dezan. Hiru fase desberdindu arren, hauek akumulatiboak direla esan beharra dago, hau da, fase bat ezingo da aurrera eraman aurrekoa guztiz ez bada garatu. Gainera, fase hauek hainbat puntutan gainjarriko dira, lan egiteko ezinbestekoa bilakatzen baita.

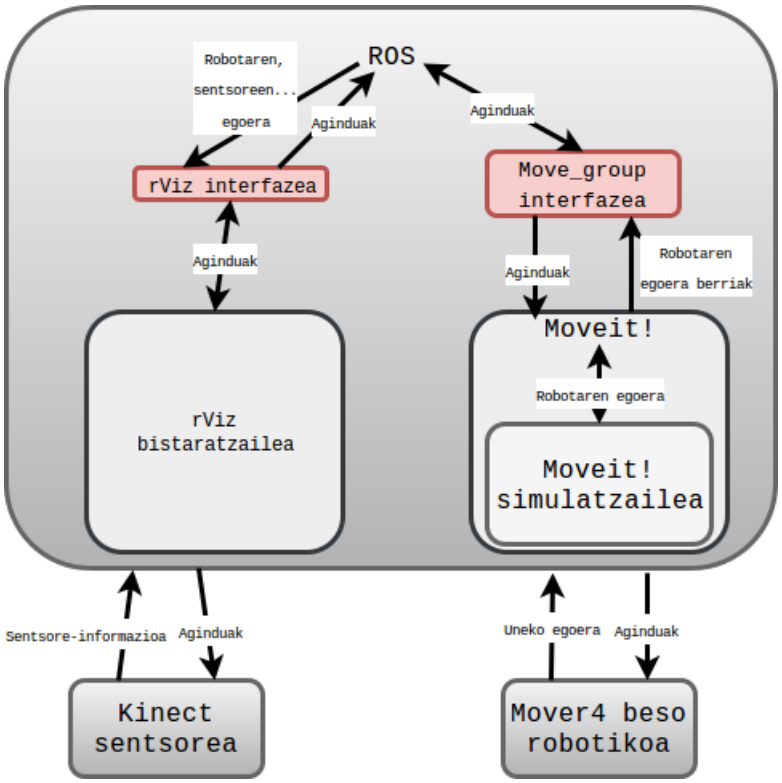
Lehenengo faseari dagokionez, robotak duen ingurunea nolakoa den zehaztu beharko dugu guztia rViz bistaratzailean bistaratu dadin. Horretarako, robotaren ingurunean azaltzen diren objektu guztiak erazagutu behar dira. Helburu hau lortzeko ROSek darabilen URDF (Universal Robot Description Format) fitxategi-formatuaz baliatuko gara. Fitxategi hauen bitartez instalazioan, hau da, robotak lan egiten duen inguruan, agertzen diren bi elementu nagusiak erazagutuko ditugu: Kinect sentsorearen euskarria eta beso robotikoa berau. Fitxategi hauetan robotaren atal eta lotura desberdinak identifikatuko ditugu, baita Kinect sentsorea eusten duen euskarria. Geroago, robotari mugitzeko aginduak emateko pluginak konfiguratu eta instalazioaren elementu bakoitzaren arteko posizio-erlazioak zehaztu beharko ditugu.

Bigarrengoz, Moveit! ingurunea erabiliz, ikusmen artifiziala gehituko diogu ingurune errealari. Ingurune honi esker, robota mugitzen den ingurunearen datuak lortuko ditugu denbora errealean. Datu horiek eskuratuta, inguruneak nahikoa informazio edukiko du beso robotikoak mugimenduak exekuta ditzan oztopoak ekidinez.

Azkenik, Kinect sentsoreak luzatzen digun informazioa filtratuko da manipulatu beharreko piezen posizioa ezagutzeko, geroago, Moveit! ingurunearen bitartez pieza horiek posizio horietatik jaso eta dagokien posizioan utzi ditzan, betiere ingurunean hautematen diren oztopo guztiak ekidinez.

Azaldutako tresna hauek guztiek elkarlanean jardungo dute, hau da, ez dira modu independentean erabiliko, izan ere bakoitzak sistema osoak funtzionatzeko beharrezkoak dira.

Tresnen arteko dependentziak eta nondik norakoak azaltzeko 1.1 irudia kontsulta dezakegu.



1.1 Irudia: *Sistema osoaren deskribapen grafikoa*

Irudian ikus dezakegunez, gure sisteman garrantzitsuena izango den tresna ROS *framework*-a izango da. Tresna honen bidez instalazioa osatzen duten elementu eta gainerako tresna guztiak beraien artean komunikatu ahal izango dute.

Hardwarearen ikuspuntutik, gure sistema honek hiru elementu nagusi izango ditu: Kinect sentsorea, beso robotikoa eta ROS exekutatzeko beharrezko konputagailua. Hadware desberdinen arteko komunikazioa gauzatzeko USB motako interfaze fisikoak erabiliko ditugu. Modu honetan, sisteman erabiltzen diren elementu guztiak ROS *framework* koordinatzailearekin komunikatuko dira.

Software aldetik, ROS en barruan inplementatuta ditugu gainerako bi tresnak: rViz bistaratzailea eta Moveit! alderantzizko zinematikarako softwarea. Irudian ikus dezakegun bezala, rViz bistaratzailea software bidezko interfaze baten bidez komunikatuko da gainerako elementuekin, ROS tartean dagoelarik. ROSek robotaren egoera eta Kinect sentsorearen informazioa luzatuko dio bistaratzaileari, beharrezko informazioa erakutsi dezan,

eta rViz-ek, trukean, ROSi bidaliko dizkio aginduak (hala nola, robotaren egoera aldatzeko edota sentsorearen informazioa ezkutatzeko).

Moveit! softwarearekin komunikatzeko, ROSek *move_group* interfazea erabiliko du. Interfaze honen bitartez ROSek Moveit! softwareari aginduak bidaliko dizkio (mugimenduak kalkulatu robota posizio jakin batetara mugitzeko, oztopoak ekiditeko...) eta honek ere ROSi bidaliko dizkio (robotaren egoera aldatzeko, bistaratzaile zein errealitatean, edota erabiltzaileari abisuak emateko).

Moveit! softwarearen barruan simulatzaile bat aurkituko dugu. Simulatzaile honek erabiliko dugun rViz bistaratzailearekin bat lan egingo du. Behin Moveit! softwareak mugimenduak kalkulatu dituenean, informazioa ROSi pasatzeaz gain, simulatzaileari ere pasako dizkio. Modu honetan, simulatzaileak mugimendu horiek simulatuko ditu eta robotaren egoera berriak bidaliko dizkio rViz bistaratzaileari, berriro ere ROS bitartekari izanik.

Erabili beharreko tresnen berritasuna dela eta, garapenean hainbat zailtasun gainditu beharko direla aurreikusten da. Zailtasun horien artean erabili begarreko tresnen arteko komunikazio eta koordinazioa, ikusmen artifizialaren erronkak eta tresnak erabili orduko ezjakintasuna daude.

Honako memorian proiektuaren nondik norakoak azaltzen dira hainbat ataletan banatuta. 2. atalean proiektuaren kudeaketari dagokion Proiektuaren Helburuen Dokumentua azaltzen da. Hurrengo hiru ataletan, (3. eta 4.eta 5. atalak) ROS framework-ari, rViz bistaratzaileari eta Moveit! inguruneari buruzko oinarrizko ideiak bilduko dira. 5. ataletik aurrera proiektuaren garapenari helduko diogu: 6. kapituluan garatutako instalazioari buruz hitzegingo da eta 7.ean, aldiz sistema horren garapena rViz bistaratzailean nola egin den ikusiko dugu. 8. atalean proiektuan alderantzizko zinematika Moveit! softwarearen bitartez nola gauzatu den adieraziko dugu eta 9.goan, garapenari buruzko azken atalean, piezen manipulazioa nola egingo den azalduko da. Azkenik, 10. kapituluan proiektua garatu ondorengo ondorioak azaltzen dira.

2. KAPITULUA

Proiektuaren Helburuen Dokumentua

Honako kapituluan proiektuaren garapenean zehar egindako sistema azalduko da. Bertan, proiektuak bete beharreko helburuetaz gain, atazen deskonposaketa eta egindako atazen zerrenda ere azalduko dira. Honez gain, egindako lan guztia modu kronologikoan azaltzen duen kronograma bat ere badago, atazak burutzeko egindako orduen estimazioarekin batera. Gainera, komunikazio, kalitate zein arriskuen plana azalduko dira, eta, eskuraketen kudeaketa eta lan-metodologia aztertu eta gero, egindako dedikazio orduen estimazioa errealitatean gertatutakoarekin konparatuko dugu, desbideraketak aztertuz.

2.1 Irismena

Proiektuaren helburu nagusia piezak manipulatzeko eta aldiberean bidean aurkitzen dituen obstakuluak gainditzen dituen beso robotiko baten kontrola garatzean datza. Beso robotiko honen kontrola ROS ingurunean garatuko da, rViz bistaratzailearen eta Moveit! ingurunearen bitartez.

Proiektuan zehar erabiliko den beso robotikoa CommonPlace Robotics etxeko Mover4 beso robotikoa izango da, Kinect sentsorearekin batera instalatuko dena UPV/EHUko Donostiako Informatika Fakultatean RSAIT taldeak duen laborategian.

Sistema guztia rViz-en simulatu ondoren, sistema errealarekin egingo da lan, portaera erreala simulatutakoarekin bat etortzen denentz aztertzeko.

Lan hau burutzeko oinarria ROS framework-a izango da. Kasu partikular honetan, Indigo

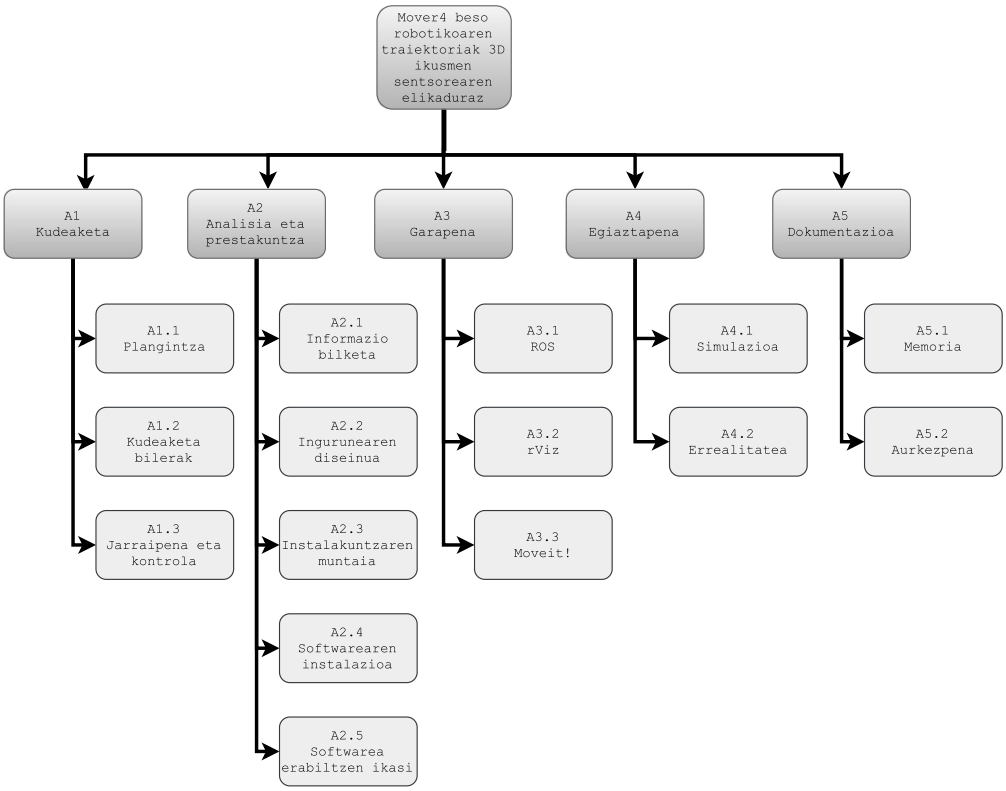
bertsioa erabiliko da Ubuntu 14.04 sistema eragilean. Erraminta honen bidez garatuko dira proiektua aurrera eramateko beharrezko fitxategi eta konfigurazio guztiak.

Simulazioari dagokionez, rViz bistaratzailearekin lan egitea aukeratu da. Honen bitartez, robotaz gain, bere ingurunea irudikatuko da, inguruan dauden objektuak barne.

Robotaren ingurunea hainbat objektuk osatuko dute. Hasteko, robota berau simulatuko dugu, horretarako ROSek eskaintzen duen URDF (Universal Robot Description File) bateragarritasuna aprobetxatuz. Fitxategi batean robotaren arkitektura eta loturak adieraziko dira. Beste URDF fitxategi batean, Kinect sentsorearen euskarria diseinatuko da simulatzailean ikus dadin. Besoaren inguruan azaltzen diren gainerako objektuak (manipulatu beharreko piezak edota obstakuluak) Kinect sentsoreak eskaintzen duen puntuen laino baten bitartez adieraziko dira simulatzailean bertan.

2.2 Lanaren deskonposaketa egitura

2.1 irudian agertzen den diagraman proiektuaren lan-paketeak ikus daitezke.



2.1 Irudia: *LDE diagrama*

2.3 Atazak

2.1 irudian agertzen diren lan-paketeei jarraituz, honakoak izan dira proiektua garatzeko landu beharrezko atazak:

A1 Kudeaketa

A1.1 Plangintza

A1.1.1 Irismenaren zehaztapena

A1.1.2 LDE diagrama sortu

A1.1.3 Atazen zehaztapena

A1.1.4 Kronograma sortu

A1.1.5 Beharrezko orduen estimazioa egin

A1.1.6 Kalitate plana zehaztu

A1.1.7 Arriskuen plana zehaztu

A1.1.8 Eskuraketen kudeaketa

A1.1.9 Lan-metodologia zehaztu

A1.2 Kudeaketa bilerak

A1.2.1 Konstituzio bilera

A1.2.2 Tarteko bilerak

A1.2.3 Itxiera bilera

A1.3 Jarraipena eta kontrola

A1.3.1 Desbideraketen kontrola burutu

A1.3.2 Desbideraketen arrazoiak aztertu

A2 Analisia eta prestakuntza

A2.1 Informazio bilketa

A2.1.1 ROS

A2.1.2 Mover4 beso robotikoa

A2.1.3 Kinect sentsorea

A2.1.4 rViz

A2.1.5 Moveit!

A4.2.1 Beso robotikoa mugitu

A4.2.2 Oztoporik gabe, besoa Moveit!-en bidez mugitu

A4.2.3 Oztopoekin, besoa Moveit!-en bidez mugitu

A4.2.4 Piezak leku desberdinetan jarri eta identifikatu

A4.2.5 Piezen manipulazioari ekin, oztoporik gabe

A4.2.6 Piezen manipulazioari ekin, oztopoekin

A5 Dokumentazioa

A5.1 Memoria idatzi

A5.2 Erabilpen gida idatzi

A5.3 Defentsarako materiala prestatu

2.4 Kronograma

2.2 irudian azaltzen da proiektuaren lan paketeak denboran zehar kokatzeko erabilgarria den kronograma. Bertan, kronologikoki ordenaturik, egindako lan bakoitza zein astetan garatuko den azaltzen da. Proiektu osoaren garapenak 26 aste emango ditu.

Aste horietatik lehenengo bostak plangintza eta informazio bilketa egiteko planifikatu direla ikus dezakegu. Honez gain, proiektua aurrera eramateko lehenengo bilerak ere egingo direla ikusten da. Ataza hauetaz gain, denbora gutxiago emango duten beste hiru ataza aurkitu ditzakegu, instalazioaren diseinua, muntaia eta proiektua martxan jartzeko softwarearen instalazioa.

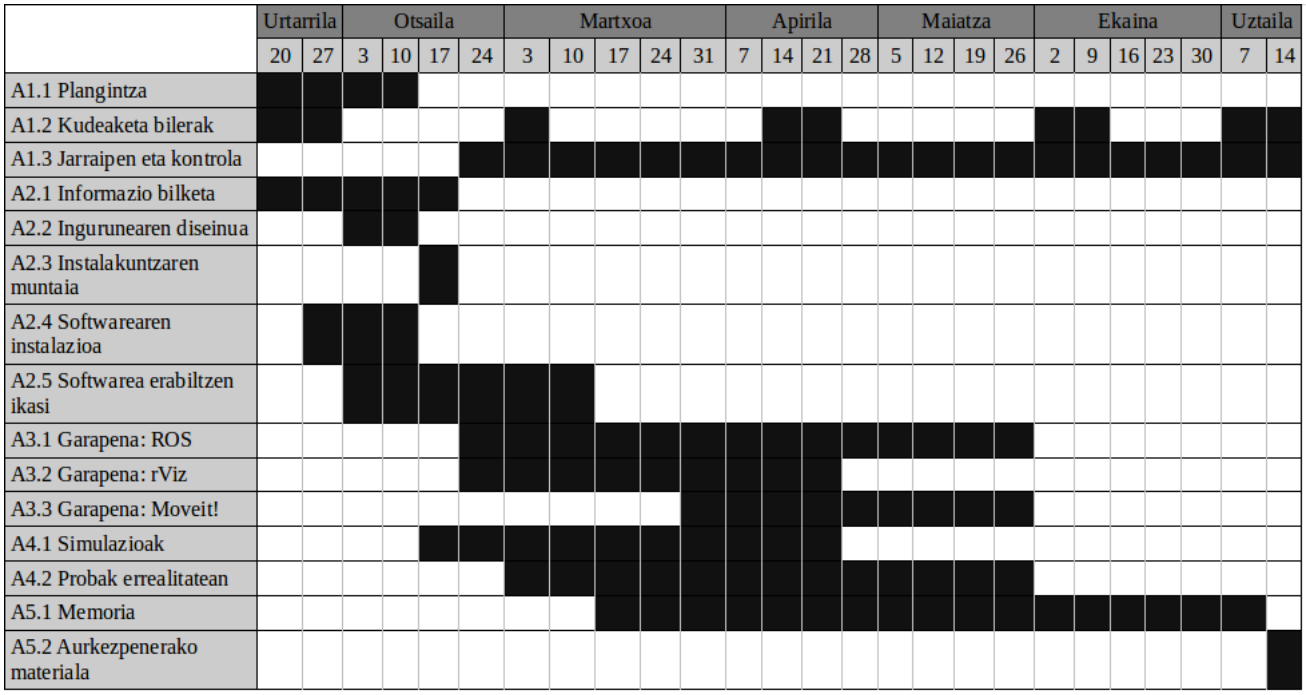
Hortik aurrerako 14 asteak proiektuan erabili beharreko softwarea erabiltzen ikasten eta proiektuaren garapenean emango dira. Aste horietan garatuko dira ROSeko fitxategiak, simulazioak egingo dira rViz bistaratzailearekin, eta Moveit!-eko konfigurazioa garatuko da. Honez gain, garatzen ari den proiektua errealitatean probatzen joango da simulazioak emaitza onak ematen dituen bitartean.

Proiektuaren garapeneko fasean memoria garatu arren, azkeneko 6 asteak eskeiniko zaizkio memoriaren garapenari modu esklusiboan. Azkeneko 3 asteetan zehar erabilpen gida eta aurkezpenerako materiala garatuko dira.

Proiektu osoan zehar, plangintza bukatu ostean, proiektuaren garapenean eta proiektuaren azkeneko asteraino jarraipen eta kontroleko prozesuei ekingo zaie. Azkenik, denboran

sakabanaturik, kudeaketa bilerak egingo dira proiektuari buruzko aldaketak zehazteko, hobekuntzak aplikatzeko edota proiektuaren kontrola egiten jarraitzeko.

Xehetasun gehiagorako, ikus 2.2 irudiko kronograma.



2.2 Irudia: *Kronograma*

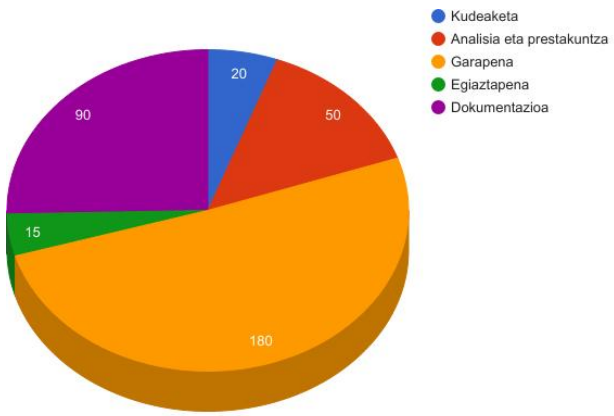
2.5 Dedikazio estimazioa

2.1 taulan agertzen dira aurretik aipatutako lan-paketeak osatzeko estimatutako ordu kopuruak. Bertan azaltzen diren datu guztiak modu grafikoan adierazten dituen sektore-grafiko bat ere sortu da, 2.3 irudian ikusgai dagoena.

Ikus daitekeen bezala, lan-pakete guztietatik ordu gehiagotan estimatu dena garapenaren fasea da. Honen ondoren, dokumentazioari, analisi eta prestakuntzari, kudeaketari, eta, azkenik, egiaztapenari dedikatu zaie ordu gehien.

Lan-paketea	Dedikazio estimazioa (orduak)
A1. Kudeaketa	20
A1.1. Plangintza	10
A1.2. Kudeaketa bilerak	3
A1.3. Jarraipena eta kontrola	7
A2. Analisia eta prestakuntza	50
A2.1. Informazio bilketa	15
A2.2. Ingurunearen diseinua	3
A2.3. Instalakuntzaren muntaia	1
A2.4. Softwarearen instalazioa	2
A2.5. Softwarea erabiltzen ikasi	29
A3. Garapena	180
A3.1. Garapena: ROS	80
A3.2. Garapena: rViz	30
A3.3. Garapena: Moveit!	70
A4. Egiaztapena	15
A4.1. Simulazioak	5
A4.2. Probak errealitatean	10
A5. Dokumentazioa	90
A5.1. Memoria	75
A5.2. Aurkezpena	10
Guztira	350

2.1 Taula: Dedikazioaren estimazioa



2.3 Irudia: *Orduen estimazioen sektore-grafikoa*

2.6 Komunikazio plana

Proiektuan zehar agertzen diren zalantza eta arazoak EHUko e-posta zerbitzuaren bidez komunikatuko zaizkio proiektuko zuzendariari ikerketa taldeko laborategian ez badago.

Proiektuaren inguruko aldaketak, hobekuntzak edota planifikazioaren aldaketak laborategian eztabaidatuko dira bileren bitartez. Bilera hauetaz aparte, proiektuaren bukaeran, itxiera bilera bat egingo da.

2.7 Kalitate plana

Honako atalean proiektuaren kalitate plana adierazten da. Bertan, proiektuan garatutako instalazioaren kalitate betekizunak agertzen dira.

Honakoak dira produktuak bete beharreko oinarrizko betekizunak:

- Robotak lan egingo duen ingurunea simulatu rViz bistaratzailean.
- Robotaren mugimenduak simulatu.
- Oztopoen detekzioa gauzatu Kinect sentsorearen bidez.
- Manipulatu beharreko piezen posizioak identifikatu Kinect sentsorearekin.
- Simulazioan beso robotikoak piezen manipulazioa egitea, inongo oztoporik jarri gabe.
- Simulazioan beso robotikoak piezen manipulazioa egitea, oztopoak jarrita.
- Beso robotikoak piezen manipulazioa egitea, inongo oztoporik jarri gabe.
- Beso robotikoak piezen manipulazioa egitea, oztopoak jarrita.

Produktuaren kalitate-dimentsioak, aldiz, honakoak dira:

- Robotak lan egingo duen ingurunea simulatzean, inguru hori errealitatera ahalik eta modu zehatzenean errepresentatzea, neurriak, koloreak, ehundurak... kontuan hartuz.

- Oztopoen detekzioa ahalik eta zehatzena izatea, beharrezko konfigurazioak eginik, oztopoak ahalik eta modu seguruenean ekiditeko.

- Kinect sentsorearen informazioa ahalik eta modu zehatzenean segmentatzea, manipulatu beharreko piezaren posizioa ondo identifikatzeko.

- Errealitate nahiz simulazioan, piezen manipulazioa modu egokian egitea, ahalik eta huts gutxien gertatzeko.

2.8 Arriskuen plana

Honako atalean, proiektuaren garapenean zehar gerta daitezkeen arazoei konponbidea ematen saiatuko gara. Arazo hauek aldez aurretik identifikatuta, errazagoa izango da aurrerantzean, arazoa gertatzen bada, honi aurre egitea eta proiektuaren garapenari ahalik eta eragin gutxien egitea.

Ondoren agertzen diren arrisku horiek guztiek ez dute proiektuaren gain eragin bera izango. Hori dela eta, eragin handien edota gertatzeko probabilitate handiena duten arazoei konponbide elaboratuenak ematen saiatuko gara.

Kontuan hartu beharrekoak dira, baita ere, garapenean zehar gerta litezkeen eta identifikatu gabe dauden arazoak. Horien eragina gutxitzeko, proiektuaren garapenean zehar denbora gehiago estimatu da arazo posible hauei aurre egiteko.

Hona hemen identifikatutako arazoak eta beraien konponbide posibleak:

1. Softwarearen matxura:
   Garapenaren fasean posiblea da egindako lan guztia ezin atzitu izatea edota beste arrazoi batzuengatik lana erabilezin bilakatzea.

   Aldiro segurtasun kopiak egitea izango da arazo honi aurre egiteko estrategia. Segurtasun kopia hauek egunero egingo dira, bi euskarri desberdin erabiliz. Lehenengoa, Google Drive zerbitzuan egongo da, bigarrena, aldiz, disko gogor eramangarri batean. Modu honetan, software matxura bat gertatuko balitz, gehienez, egun bateko lana galtzea suposatuko luke. Honako arazoa sortuko balitz, azkenengo bertsioa bulegoko ordenagailu batean edota proiektua garatzeko erabiliko den eramangarrian deskargatzea izango da konponbidea.

2. Hardwarearen matxura:
   Hardwarean gerta litezkeen matxurak kontuan hartzea garrantzitsua da. Kasu honetan, hiru hardware garrantzitsuek hartzen dute parte garatu beharreko instalazioan: Kinect sentsorea, beso robotikoa eta erabiltzen den eramangarria.

   Kinect sentsorearen matxuraren kasuan, ahalik eta denbora gutxienean beste bat lortzea izango litzateke konponbidea. RSAITek baditu hainbat kinect erabilgarri, matxurei aurre egiteko lain.

   Beso robotikoaren kasuan, matxura baten aurrean denbora laburrenean arazoak ematen dituen zatiaren desmuntaia egingo da ahal bada matxura konpontzeko. Ezinezkoa bada, pieza horren eskaera egiten saiatuko gara. Hori ere (denbora faltagatik) ez bada posible, ahalik eta modu egokienean garatu beharko da proiektua robotaren pieza hori kontuan hartu gabe.

   Eramangarriaren kasuan, aldiz, ikerketa taldeko bulegoko beste ordenagailu batean proiektuaren azkeneko bertsioa deskargatu eta bertan garapenarekin jarraitzea izango litzateke soluzioa. Eramangarria konpondu ostean, proiektua bertan garatzen jarraituko da.

3. Denbora:
   Proiektu luze denetan bezala, denbora-falta arazorik handienetako bat bilakatu daiteke. Denbora-falta honek proiektuaren helburu guztiak ez betetzea edota kalitatearen jaitsiera suposa dezake.

   Arazo larri honi aurre egiteko, plangintzan proiektuaren garapenaren eta dokumentazioaren denbora-estimazioen ordu kopuruak igo dira. Estimazioak igo arren ere denbora-falta existituko balitz, garrantzi gutxien duten helburuak alde batera utziko lirateke, eta proiektuaren kalitatea ahalik eta gutxien jaisten saiatuko ginateke.

4. Erraminten zailtasuna:
   Proiektua garatzeko erraminta guztiak lehenengo aldiz erabiltzen direnez, hauek erabiltzerako orduan esperientzia eza horrek denbora galera handiak ekar ditzake.

   Erramintak erabiltzerako orduan ahalik eta denbora gutxien galtzeko esperientzia arazoak direla eta, hauek erabiltzean sortzen diren zalantza guztiak proiektuko zuzendariari eta bere kideei galdetuko zaizkie. Modu honetan, erraminta erabiltzen ikasteko prozesua errazten da, eta denbora galerak gutxitzen dira.

2.9 Eskuraketen kudeaketa

Proiektu honetako eskuraketa guztiak Internet bitartez egingo dira. Honen bitartez lortuko dira proiektua garatzeko beharrezkoak diren plugin edota ROS paketeak. Eskuraketa hauek proiektuaren garapenean zehar adostutakoaren araberakoak izango dira. Baliteke garapen fase honetan hasieran planifikatutakoak baino eskuraketa gehiago zein gutxiago behar izatea, baina guztiak aurretik aipatutako teknikarekin gauzatuko dira.

2.10 Lan metodologia

Honako proiektu hau gauzatzeko puntu hauetan oinarritzen den lan metodologia ezarriko da:

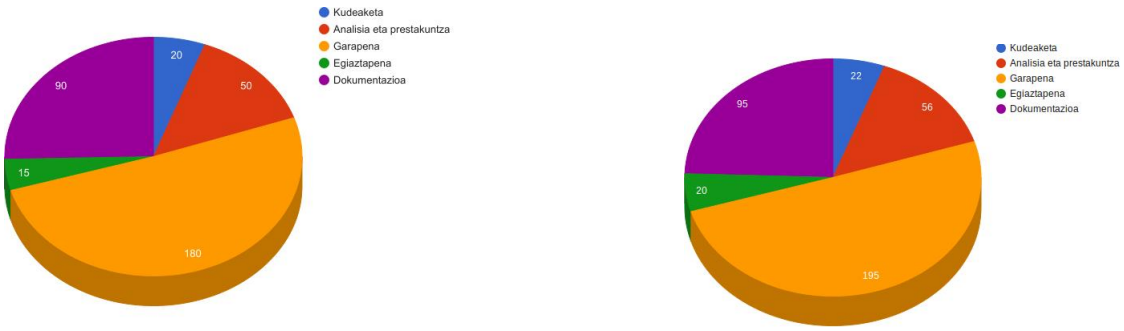
- Hasieratik, beso robotikoarekin egin beharreko proba guztiak Donostiako UPV/EHUko Informatika Fakultatean RSAIT ikerketa taldeak 3. solairuan duen laborategian egingo dira.
- Behin probak amaituta, dokumentazioaren fasea ikasleak garatuko du bere kasa, noizbehinka zuzendariari dokumentazioaren azkeneko zatiak bidaliz eta aldaketak eginez.
- Beste ikasgai batzuen klaseek irauten duten bitartean (urtarrila-martxoa hasiera) ahal denean proiektuari egunero 3 orduko dedikazioa emango zaio.
- Ikasgaiak bukatzean, egunero proiektuari gutxienez 4 orduko dedikazioa eskeiniko zaio.
- Noizbehinka, beharrezkoa den bakoitzean zuzendariarekin bilerak egingo dira. Bilera hauetan proiektuaren jarraipena egingo da, eta aldaketak edota hobekuntzak planteatuko dira.

2.11 Denbora erreala

Atal honetan 2.5 atalean estimatutako denborak errealitatean gertatutakoarekin alderatuko ditugu. Errealitatean proiektua garatzeko erabilitako denbora, 2.2 taulan ikus daiteke.

Lan-paketea	Dedikazio estimazioa (orduak)	Benetako dedikazioa (orduak)
A1. Kudeaketa	20	22
A1.1. Plangintza	10	12
A1.2. Kudeaketa bilerak	3	3
A1.3. Jarraipena eta kontrola	7	7
A2. Analisia eta prestakuntza	50	56
A2.1. Informazio bilketa	15	20
A2.2. Ingurunearen diseinua	3	3
A2.3. Instalakuntzaren muntaia	1	1
A2.4. Softwarearen instalazioa	2	3
A2.5. Softwarea erabiltzen ikasi	29	29
A3. Garapena	180	195
A3.1. Garapena: ROS	90	100
A3.2. Garapena: rViz	30	20
A3.3. Garapena: Moveit!	70	75
A4. Egiaztapena	15	20
A4.1. Simulazioak	5	7
A4.2. Probak errealitatean	10	13
A5. Dokumentazioa	90	95
A5.1. Memoria	75	75
A5.2. Aurkezpena	10	15
Guztira	350	383

2.2 Taula: Dedikazioaren estimazioa eta benetako dedikazioa alderatuta



2.4 Irudia: *Estimatutako orduen eta errealitateko orduen sektore-grafikoen konparaketa*

2.2 taulan eta 2.4 irudian ikus daitekeen bezala, lan-pakete bakoitza exekutatzean desbiderapenak egon dira. Honakoak izan dira desbiderapenak:

- A1. Kudeaketa:
  Honako lan-paketean 2 orduko desbiderapena gertatu da A1.1 lan-paketean, plangintza egitean, hain zuzen ere 2 ordu gehiago inbertitu baitira.

- A2. Analisia eta prestakuntza:
  Honakoan 6 orduko desbiderapena gertatu da. A2.1 (informazio bilketa) eta A2.4 (softwarearen instalazioa) lan paketeek 5 eta 1 orduko desbiderapenak jasan dituzte, hurrenez hurren.

- A3. Garapena:
  Honakoa izan da desbiderapen gehien jasan duen lan-paketea, 15 orduko desbiderapenarekin. Lan-pakete honetako pakete guztiek jasan dituzte desbiderapenak. A3.1 eta A3.3 lan-paketeek denbora gehiago eskatu duten bitartean, A3.2 paketeak 10 ordu gutxiago eskatu ditu.

- A4. Egiaztapena:
  Lan-pakete honetan 5 orduko desbiderapena egon da simulazioek 2 ordu gehiago behar izan dituztelako eta errealitateko probek, aldiz, 3 ordu gehiago.

- A5. Dokumentazioa:
  Lan-pakete honek, aurrekoak bezala, 5 ordu gehiago eskatu ditu bukatzeko. Aurkezpenerako materiala sortzeko lan-paketeak, A5.2 paketeak, alegia, 5 ordu gehiago behar izan baititu.

Guztira, proiektuak 33 orduko desbiderapena izan du, 33 ordu gehiago inbertitu behar izan dira proiektua bukatzeko.

3. KAPITULUA

ROS: Robot Operating System/Robot Open Source

3.1 Sarrera

Honako atalean proiektua garatzeko oinarria bilakatu den ROS framework-ari buruzko oinarrizko kontzeptuak azalduko dira, proiektuaren garapena ulertzeko. Horretarako, ROSen funtzionamenduaren nondik norakoa azalduko da, oinarrizko kontzeptuetatik hasiz, nodoen arteko komunikazioa aztertuz eta, azkenik, proiektuan eragina duten bestelako kontzeptu eta funtzionalitate batzuk azalduz.

ROS, egun, robotak kontrolatzeko UNIX sistemetan erabiltzen den framework-a da. Framework hau robotikako erronkak leuntzeko plataforma zabala da, kode irekikoa eta komunitate handi batek egindakoa. Hori dela eta, merkatuko hainbat robotentzako liburutegiak, erraminta eta kode asko ditu eskuragai.

Sistema hau grafo arkitektura batean oinarrituta dago. Grafo hau hainbat nodok (exekutagarriek) osatzen dute. Nodoa, robot bat edota instalazio bat osatzen duen zati bat izango da, zeregin konkretu bat duena, prozesu bat izango balitz bezala. Grafoa osatzen duten nodoen arteko komunikazioa P2P (Peer to Peer) motakoa da, nodo bat beste edozein nodorekin komunikatu daiteke zuzenean.

Honez gain, ROSek beste ezaugarri interesgarri bat du: programazio-lengoaiarekiko independentea da eta, egun, C++, Python, Octave, Java eta bestelako programazio-lengoaiekin lan egiteko gai da. *Framework*-ari buruzko informazio gehiagorako bere webgune ofizialera jo daiteke [1].

3.2 ROSen kontzeptu nagusiak

Atal honetan ROS framework-a osatzen duen fitxategi-sistemaren eta bere barneko elementuek osatzen duten arkitekturaren nondik norakoak azalduko dira.

3.2.1 Fitxategi-sistema

ROSen fitxategi-sistema hainbat elementu desberdinek osatu dezaketen arren, hona hemen gure proiektuan garrantzitsuak bilakatu direnak:

- ROS paketeak:
  Paketea garatzen den guztia ordenatzeko unitate nagusia da. ROS pakete hauek hainbat elementu desberdinek osa dezakete, hona hemen nagusienak:
  - Programazio lengoia batean kodetutako programak.
  - Exekutagarriak.
  - Konfigurazio fitxategiak.
  - Plugin-ak.
  - Datu-multzoak.

  Pakete-sistema honen bidez erraza bilakatzen da berauen berrerabilpena egitea, edota pakete batzuk beste batzuen zati batzuk edo pakete osoak bere baitan hartzea, behar dutena erabil dezaten.

- Pakete-manifestuak (package.xml):

  Manifestuek paketeen informazioa gordetzen dute. Informazio hau mota askotarikoa izan daiteke, eta paketeak behar duenaren araberakoa izango da informazio hau. Informazio-moten artean, honakoak aurki ditzakegu:
  - Paketearen izena
  - Paketearen bertsioa
  - Mantentzea egiten duenaren izena
  - Paketeak beharrezkoak dituen liburutegiak

  - Paketaren dependentziak

  Fitxategi hau ondo definitu ezean paketeak ez du ongi funtzionatuko, bertako metainformazioa oso garrantzitsua baita berau eraikitzeko eta exekutatzeko orduan.

- Launch fitxategiak (.launch):
  Fitxategi hauen bidez hainbat nodo jar ditzakegu martxan (hauen bitartez paketetan dauden hainbat programa exekutatu ditzakegu batera). Honez gain, exekuzioan zehar erabili nahi ditugun parametroak ere finka ditzakegu bertan. Laburbilduz, launch fitxategiak nodoak martxan jartzeko erraztasunak ematen ditu.

3.2.2 Arkitektura

ROS ulertzeko beste kontzeptuen artean framework-a osatzen duten elementu nagusiak identifikatu behar dira. Honakoak dira elementu nagusiak eta beraien eginkizuna:

- ROS master-a:
  Framework-a bezero/zerbitzari ereduan oinarrituta dagoenez, honek ere zerbitzari bat du: ROS master-a. Honek nodoak exekutatzen uzten du, baita nodoak beraien artean aurkitu eta komunikatzea. Nodo honek, gainera, orain aztertuko dugun parametro-zerbitzaria jarriko du martxan.

- Nodoak:
  Nodoak prozesuak dira, robot edo gailu baten driver-a zein bezero-programak.

  Adibide gisa gure proiektua hartuz, adibidez, Kinect sentsorea nodo batez osatuta dago. Nodo honek Kinect-aren kontrola ematen du. Beste nodo bat, adibidez, besoaren kontrolaz arduratzen da, aginduak bidaltzen dizkio robotari berau mugi dadin. Hirugarren nodo batek robotaren ingurunearen bistaratzailea martxan jartzen du, eta abar.

  Nodoek, noski, elkarlana egin dezakete, eta horretarako komunikatu egiten dira. Komunikazioaren arloa hurrengo atalean landuko dugu.

- Parametro-zerbitzaria:

  Parametro-zerbitzaria ezinbestekoa da nodoek beraien aldagaiak eta parametroak gordetzeko. Honi esker, gainera, hainbat nodok beraien parametroak partekatu ditzakete, elkarlanean lan egin dezaten.

3.3 Komunikazioa

ROSeko oinarrizko kontzeptua komunikazioarena da, nodoek elkarrekin lan egiteko beharrezkoa dute komunikazioa. Lehen esan bezala, komunikazio hau P2P (Peer to Peer) motakoa da, hau da, nodoen sarean nodo guztiak dira bezeroak. Bezero hauek elkarrekin komunika daitezke zuzenean beraien arteko mezuak trukatu edo zerbitzuak emateko.
ROSen, komunikazioa bi modutara gauzatu daiteke: mezu bidez (komunikazio asinkronoa) edota zerbitzu bidez (komunikazio sinkronoa).

3.3.1 Komunikazioa Mezuen bidez

Lehenengo komunikazio modua mezuen bidezkoa da. Komunikazio modu honetan nodoek beraien artean mezuak trukatzen dituzte. Mezu hauek formatu desberdinak izan ditzakete. Mezuen formatua orokorra izan daiteke (*String*-ak, zenbakiak...), aldez aurretik definitutakoak (*sensor_msgs* motako mezuak, adibidez) edota garatzaileak berak sortutakoak.

Mezu hauek beraien artean trukatzeko derrigorrezkoa bihurtzen da postontzi baten existentzia. ROSen, postontzi hauei *topic* izena eman zaie. Mezuak trukatzeko, nodo batek mezu horiek argitaratuko dituela adierazi beharko du (*Publisher*-a izan beharko du). Mezu horiek jaso nahi dituzten nodoek *topic* horretara harpidetu beharko dute (*Subscriber*-ak izango dira). *Topic* berean hainbat argitarazle eta hainbat harpidedun egon daitezke.

3.3.2 Komunikazioa zerbitzuen bidez

ROSek eskaintzen duen bigarren motako komunikazioa zerbitzuen bidezkoa da. Kasu honetan, nodoak eskaera/erantzun bidezko eredua erabiliz komunikatzen dira. Komunikazio modu honek bi norabideko komunikazioa ahalbidetzen du, dagoeneko ez da beharrezkoa izango inongo *topic*-etan harpidedun izatea, oraingoan nodo batek zerbitzu bat eskainiko du, gainontzeko nodoek eskaerak luzatzen dituzten heinean, bezero/zerbitzari ereduan bezala. Beraz, komunikazio sinkronoa bilakatzen da komunikazio modu hau.

3.4 Beste kontzeptu batzuk

ROSeko oinarrizko kontzeptuak azaldu ondoren, ROSeko nukleotik at eraiki daitezkeen sistema konplexuagoak ditugu. ROSek oraintxe azaldutako komunikazio moduak eta arkitektura eskeini arren, ez du inondik inora hauek nola erabili behar diren adierazten. Hori dela eta, oinarrizko sistema honen gainean goi-mailako sistemak eraiki daitezke. Honako atalean gure proiektuan parte hartzen duten goi-mailako sistema horiek aztertuko ditugu.

3.4.1 Posizio-transformatuak: *tf*

Gure proiektuaren helburua lortzeko, hau da, piezak manipulatzeko ezinbestekoa dugu jakitea beso robotikoa zein posiziotan dagoen, Kinect sentsorearen posizioa, altuera... laburbilduz, inguruneko elementu bakoitza non dagoen eta beraien arteko erlazioa jakin beharra dugu.

Inguruneko objektu bakoitzak bere erreferentzia-sistema propioa edukiko du; adibidez, Kinect sentsoreak pieza bere buruarekiko non dagoen jakingo du, baina posizio hori ez da bat etorriko robotaren erreferentzia-sistemarekin.

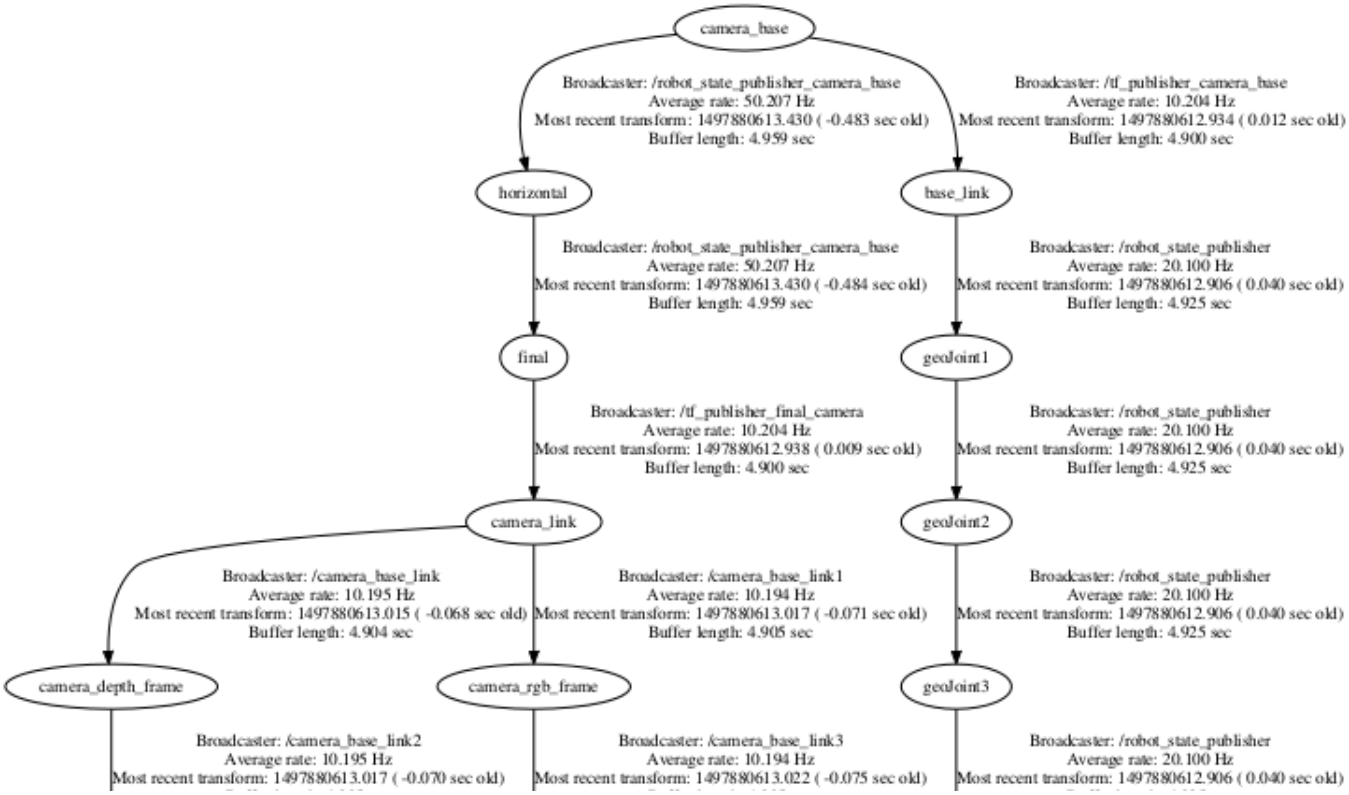
Hori dela eta, derrigorrezkoa da inguruneko elementuak erreferentzia-sistema amankomun batean biltzea. Hori lortzeko, beharrezkoa da *tf* liburutegia erabiltzea.

Liburutegi hau erreferentzia-sistemen arteko bihurketak egiteko gai da. Horretarako, sistema osatzen duten objektu guztien erreferentzia-sistemak tf_tree deritzon zuhaitz batean biltzen dira. Honen bidez, inguruneko objektu guztien posizio- eta errotazio-informazioa gordeko da, beraien arteko transformazioak burutzeko.

Gure proiektu honetan, hainbat erreferentzia-sistema ditugunez, beharrezkoa izango da objektu bakoitzaren transformatua konfiguratu eta argitaratzea. Lan hori errazteko, *robot_state_publisher* deritzon paketea dugu. Honi esker objektuen egoerak publikatuko dira. Pakete honekin, adibidez, instalazioan parte hartzen duten elementuen arteko erlazioak grafikoki ikus ditzakegu, transformazioak egiterakoan beraien arteko dependentziak aztertzeko, komando honen bitartez:

```
rosrun tf view_frames
```

Komando honekin pdf fitxategi bat sortuko da, eta aurretik aipatutakoa grafo baten bidez azalduko zaigu (ikus: 3.1 irudia).



3.1 Irudia: *instalazioko elementuen tf_tree-aren zati bat*

3.4.2 Roboten ereduak: URDF (Unified Robot Description Format)

Honako fitxategiek roboten edota ingurunea osatzen duten elementuak definitzeko gaitasuna ematen dute. Honez gain, aurretik aipatutako posizio transformatuak kudeatzeko ezinbestekoak diren fitxategiak dira. Hauek XML formatua jarraitzen duten fitxategiak dira, hau da, etiketen bitartez definitzen dira robot baten zatiak eta beraien arteko erlazioak.

Konfigurazio-fitxategi hauen bitartez instalazioa osatzen duten elementuak simulatu eta bistaratu ahal izango ditugu.

3.4.3 Moveit!

Moveit! softwarea robotetan alderantzizko zinematika aplikatzea ahalbidetzen duen ingurunea da. ROS pakete moduan antolatzen da eta gure proiektuan robota posizio batetik bestera mugitzeko aginduak emango dizkio.

Ingurune honek, gainera, kanpoko sentsoreen informazioa erabiltzen uzten digu, adibidez, ikusmen sentsoreak. Honen bitartez Moveit!-ek mugimenduak planifikatzean obstakuluak ekidingo ditu mugitzen den bitartean.

3.4.4 Semantic Description Robot Format: SDRF

Honako konfigurazio-fitxategiek ere robot baten morfologia zehazteko balio dute. Hauek, URDF fitxategien luzapen gisa har daitezke, URDF fitxategiek definitzen uzten ez dituzten elementuak definitzea ahalbidetzen baitute, hala nola, robot bateko lotura desberdinak biltzen dituzten lotura-taldeak, mugimendu espezifikoak soilik lotura-multzo bati aplikatzeko definitutakoak.

Honako konfigurazio-fitxategiak Moveit! paketeak erabiltzen ditu roboten mugimenduak planifikatzeko orduan.

4. KAPITULUA

rViz

4.1 Sarrera

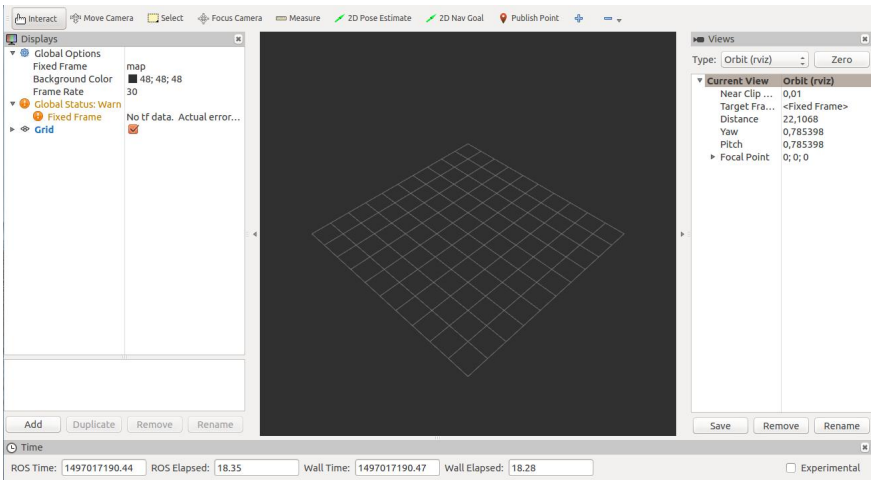
Robot baten funtzionamendua aztertzeko, diseinatutako algoritmoek, mugimenduek, eginkizunek... ongi funtzionatzen dutela ziurtatzeko eta batez ere, inguruko pertsonen eta robot beraren segurtasuna bermatzeko nahitaezkoa izaten da aldez aurretik funtzionamendua bistaratzaileen eta simulatzaileen bitartez aztertzea . Proiektu honetan parte hartzen duten elementuak bistaratzeko rViz softwarea erabiliko dugu.

rViz ROS-en gainean exekuta daitekeen 3D bistaratzaile bat da. Bertan, hainbat sentsoreen informazioa bistaratzeaz gain robotaren uneko egoera ikus daiteke. Honez gain, ingurunea osatzen duten objektuak erazagutu eta bisualizatu daitezkeenez, elementu guztien arteko posizio-transformatuak aplikatuz errealitateko instalazioa modu fidagarri batean bistaratu daiteke.

Behin instalazioa bisualizatuta, errealitatean egin daitezkeen proba guztiak egiteko gai izango gara. Bertan Kinect-aren informazioa ikus dezakegu, beso robotikoaren mugimenduak bisualizatuko ditugu, piezen manipulazioa simulatu dezakegu eta, laburbilduz, errealitatean egin nahi ditugun proba guztiak egingo ditugu ekipoaren zein inguruko pertsonen segurtasuna bermatuz.

Bisualizatzaile honek, gainera, hainbat plugin desberdinekin egin dezake lan, gure kasuan erabiliko dugun Moveit! alderantzizko zinematika aplikatzeko erabiliko dugun ingurunea, esaterako. Simulatzaile honekin plugin-ak kalkulatutako mugimenduak simulazioan ikus ditzakegu, baita berauk Kinect-aren informazioarekin somatutako oztopoak ere. Informazio gehiago bere webgune ofizialean aurki dezakegu [2].

Proiektu hau garatzerakoan, rViz bisualizatzailearen 1.11.15 bertsioa erabili da ROS Indigo bertsioaren gainean, Ubuntu 14.04 sistema eragilea erabiliz. 4.1 irudian bisualizatzaile hutsa exekutatzean azaltzen den interfaze grafikoa dugu ikusgai.



4.1 Irudia: *rViz simulatzailearen interfaze grafikoa*

Hurrengo atalean bisualizatzailean bisualiza daitezkeen elementu batzuk aztertuko ditugu, gure proiektuan behar izan ditugunak.

4.2 Robot ereduak: URDF

Bisualizatzailean ingurunea eta robota(k) simulatzeko lehenengo pausua hauen ereduen deskribapenak sortzea da. Deskribapen hauek URDF fitxategien bidez egiten dira. XML formatua jarraitzen duten fitxategi hauetan robota edota inguruneko objektuen loturak, ardatzak, motoreak... etiketen bidez erazagutu beharko dira.

Proiektu honetan bi motatako objektuak erazagutu behar izan ditugu: dinamikoak eta estatikoak. Bi objektu moten deskribapenek eredu bera erabili arren, hurrengo bi azpiataletan hauen erazagupena egiteko oinarrizko etiketak azalduko ditugu.

4.2.1 Objektu dinamikoak

Objektu dinamikoak simulazioan eta errealitatean mugitzeko gaitasuna duten objektuak dira. Objektu hauen artean robot mugikorrak edota beso robotikoak ditugu, proiektuan erabiltzen denaren antzekoa.

Objektu mota hauek erazagutzerakoan, URDF fitxategia sortzerakoan, bi etiketa dira nagusienak: <*link*> eta <*joint*> etiketak, hain zuzen ere. Etiketa hauen bitartez robot bat osatzen duten ardatzak eta beraien arteko loturak defini daitezke, hurrenez hurren. Hauen definizioa modu askotara egin daitekeen arren, robot baten ardatzak definitzeko modu generiko bat azaltzen da 4.1 adibidean.

4.1 Adibidea: Robot baten ardatzen definizio adibide bat

```
<link name="base_link">
    <visual>
    <origin rpy="0 0 0" xyz="0 0 0.0"/>
      <geometry>
    <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint0.obj" scale="0.001 0.001
     0.001"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 0.8"/>
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint0Coll.obj" scale="0.001
     0.001 0.001"/>
      </geometry>
    </collision>
  </link>
```

Ikus dezakegun bezala, ardatz bakoitzari izena esleitu diezaiokegu <*link name*> etiketaren bidez. Honez gain, rViz-eko munduko erreferentzia-sistemako zein puntutan aurkitzen diren defini dezakegu <*origin*> etiketekin. Ehundurak definitzeko *.obj* luzapena duten fitxategiak ezarri ditzakegu. Hori gutxi izango balitz ere, geroago Moveit! pluginarekin erabiltzeko kolisioa sor dezaketen ardatzak defini ditzakegu <*collision*> etiketaren bidez.

Ardatzak definituta daudela, hauen arteko loturak definitu beharko ditugu. Lotura hauek estatikoak ala dinamikoak (motorizatuak) izan daitezke. 4.2 adibidean lotura dinamiko bat definitzen dugu.

4.2 Adibidea: Lotura dinamiko baten definizioa

```
<joint name="Joint0" type="revolute">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="geoJoint1"/>
    <origin rpy="0 0 0" xyz="0 0 0.158"/>
    <limit effort="100" lower="-1" upper="2" velocity="30" />
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
```

Aurreko kasuan bezala, honakoan ere lotura bakoitza izen batekin identifikatu dezakegu <*joint name*> etiketaren bitartez. Zein motatako lotura *type* argumentuarekin definitu beharko dugu. URDF fitxategietan hainbat lotura mota definitu daitezkeen arren, hiru dira nagusienak: *fixed*, *revolute* eta *continuous*. *Fixed* motako loturak estatikoak dira, ez dute mugitzeko ahalmenik. *revolute* motakoek, aldiz, dinamikoak dira, baino mugitzeko ahalmen mugatua dute, serboekin mugitzen diren loturak definitzeko erabiltzen dira. *continuous* motakoak motore elektrikoen bitartez kontrolatzen diren loturekin erabiltzen da, motore hauek ez dute inongo mugarik mugimenduan. *revolute* motako loturetan hainbat muga desberdin erazagutu daitezke. Kasu honetan esfortzu, posizio eta abiadura mugak ezarri ditugu <*limit*> etiketaren *effort*, *upper*, *lower* eta *velocity* argumentuen bidez.

Lotura bat definitzerakoan zein bi ardatzen arteko lotura den adierazi beharko da <*parent link*> eta <*child link*> etiketen bitartez. Kasu honetan *base_link* eta *geoJoint1* ardatzen arteko *revolute* motako lotura bat sortu da.

Aurreko kasuan bezala, loturaren posizioa adierazi daiteke rViz munduko erreferentzia-sistemarekiko. Honez gain fisikarekin erlazionatutako hainbat ezaugarri definitu ditzakegu, hala nola marruskadura *friction* argumentuarekin edota amortiguazioa *damping* argumentuaren bidez.

4.2.2 Objektu estatikoak

Modu berean, inongo mugimendurik ez duten objektuak ere defini daitezke. Gure kasuan, Kinect sentsorearen euskarria diseinatu behar izan dugu modu honetan. Etiketa berak erabiliz, modu generikoan, honela defini daitezke objektu estatikoak.

Hasteko, objektuaren materiala defini dezakegu <*material*> etiketaren bitartez (ikus; 4.3 adibidea). Kasu honetan, *grey* izeneko materiala sortu da, *rgba* kolore-sistemaren bitartez kolore grisa esleituz.

4.3 Adibidea: Material baten definizioa

```
<material name="grey">
    <color rgba=".60 .60 .60 1"/>
  </material>
```

Honez gain, lehenago egin bezala, objektuaren zati desberdinak definituko ditugu <*link*> etiketaren bitartez, 4.4 adibidean agertzen den bezala.

4.4 Adibidea: link baten erazagupena

```
<link name="camera_base">
    <visual>
      <geometry>
        <box size="1 .03 .03"/>
      </geometry>
      <origin rpy="1 1.57075 0" xyz="0 0 0.5"/>
      <material name="grey"/>
    </visual>
  </link>
```

Honakoan ere rViz munduko erreferentzia-sisteman objektua non dagoen definitu dugu, baita bere materiala ere.

Azkenik, objektuaren atal bakoitzaren arteko loturak definitu behar ditugu <*joint*> etiketaren bidez, 4.5 adibidean bezala.

4.5 Adibidea: Robot baten ardatzen arteko loturen definizioa

```
<joint name="base_to_horizontal" type="fixed">
    <parent link="camera_base"/>
    <child link="horizontal"/>
    <origin xyz="0 0 0"/>
  </joint>
```

4.3 Posizio-transformatuak

Behin instalazioa osatzen duten objektu dinamiko zein estatiko guztiak definituta, beraien arteko posizio-erlazioak definitu beharko ditugu. Hau egiteko, lehenago aipatutako *tf* paketea erabiliko dugu.

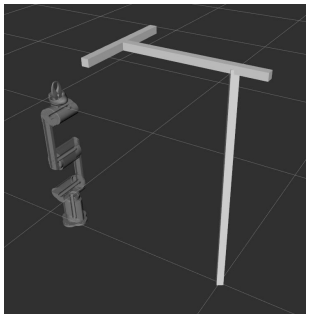
Pakete honek hainbat erreferentzi-sistemen arteko erlazioak eta transformazioak burutzea ahalbidetzen du. Hurrengo kapituluetan sakonduko dugu gai hau.

4.4 Ingurunearen simulazioa

Objektu guztiak definituta eta beraien arteko posizio-erlazioak aplikatuta, dagoeneko simula dezakegu instalazioa. Hori lortzeko, nahikoa dugu rViz simulatzailean ikusi nahi dugun objektu bakoitzeko *robotModel* deritzon osagaiak gehitzea.

Hori egin ondoren, robotaren eta euskarriaren modeloak beraien posizio transformatuekin batera exekutatu behar dira, bistaratzailean agertu daitezen.

Ondoren, bistaratzailean diseinatutako ingurunea azalduko da (ikus: 4.2 irudia).



4.2 Irudia: *Diseinatutako ingurunea rViz bistaratzailean*

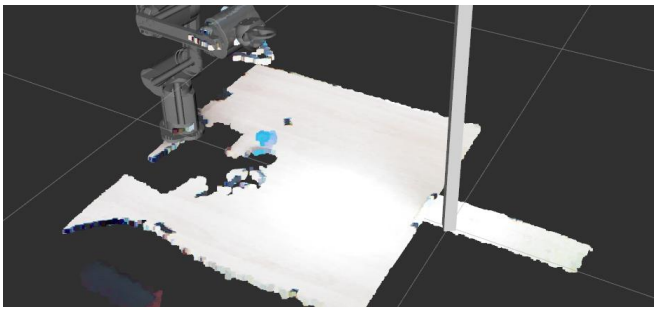
4.5 Sentsoreen informazioaren bistaratzea

Inguruneko objektuez gain, rViz-ek hainbat sentsore desberdinen informazioa bistaratu dezake, hala nola, gure proiektuan erabiliko dugun 3D kameraren informazioa. Honako sentsoreen bitartez robotaren ingurunearen aldaketak denbora errealean ikus ditzakegu, eta gure proiektuan oztopoak ekiditeko eta piezak detektatzeko erabiliko ditugu.

Aurretik aipatutako Kinect sentsorearen informazioa bistaratzeko, hasteko, rViz bisualizadorean *pointcloud2* motako elementu berri bat sortu beharko dugu. Elementu honek 3D kamerek lortzen duten informazioa ikustea ahalbidetzen du, bistaratzailean XYZ koordinatuetan eta RGB eskalako puntu-multzoa erakutsiz.

Honez gain Kinect sentsorea martxan jartzen duen *.launch* fitxategia exekutatu behar da, 3D kamera martxan jartzeko.

Hori guztia egin ondoren, bisualizadorean Kinect-aren informazioa ikusteko gai izango gara (ikus: 4.3 irudia).



4.3 Irudia: *Kinect-aren informazioa rViz bisualizadorean*

5. KAPITULUA

Moveit!

5.1 Sarrera

Robot baten mugimenduak modu efektibo eta seguruan exekutatzea ataza konplikatua bihur daiteke ingurune konplexuetan lan egiten bada. Batzuetan, gainera, zaila egiten da robot bat posizio jakin batetara eramatea zinematika hutsa erabiliz, batez ere elementu edota ardatz askoko robotetan. Arazo honen soluzio bilakatzen da alderantzizko zinematika erabiltzea.

Alderantzizko zinematika zinematika hutsa baino askoz konplexuagoa izan arren, robot bat posizio jakin batera mugitzeko metodorik eraginkorrena da. Teknika hau erabiliz, posizio horretara mugitzeko hainbat mugimendu-kolekzioen artean azkarrena, eraginkorrena edo seguruena dena exekutatu dezakegu. Teknika hau gure proiektuan aplikatzeko, Moveit! softwarea erabiliko dugu.

Moveit! ROS gainean exekuta daitekeen traiektoriak planifikatzeko tresna bat da. Tresna honek hainbat motako roboten (beso robotikoak, robot humanoideak, robot mugikorrak...) mugimenduak alderantzizko zinematika erabiliz kalkulatzeko balio du. Mugimendu hauek lortzeko robotaren atal bakoitza nola mugitu behar den kalkulatzeaz gain, mugimendu hori bisualizatzaile batean erakusteko eta robot errealari aginduak emateko gai da.

Hurrengo ataletan gure proiektua ulertzeko beharrezkoak diren Moveit!-eko kontzeptu nagusiak azalduko dira, software honen nondik norakoak uler daitezen. Informazio gehiago

bere webgune ofizialean aurki dezakegu [3].

5.2 Konfigurazio-fitxategiak

Moveit! robot batekin erabiltzeko orduan hainbat konfigurazio-fitxategi sortzea beharrezkoa da. Konfigurazio-fitxategi hauek modu arinago batean sortzeko Moveit!-ek laguntzaile bat dakar. Laguntzaile honen bitartez honako konfigurazio-fitxategiak sortu daitezke:

- SRDF fitxategia:
  SRDF (Semantic Robot Description Format), URDF fitxategiek bezala, XML formatukoak diren eta robot baten arkitektura definitzeko balio duten fitxategiak dira. URDF fitxategien bidez definitzea ezinezkoa den elementu batzuk erazagutzeko erabiltzen da. Elementu horien artean bukaerako eragingailuak (robot baten pintzak, eskuak...) eta ardatz-taldeak (robot baten hainbat lotura biltzen dituen egitura bat) daude. Ardatz-talde hauek mugimendu espezifikoak ardatz-talde jakin batek exekutatzea ahalbidetzen du, honek planifikatzerako orduan askatasun gehiago ematen duelarik. Ardatz-talde baten definizioa 5.1 adibidean aurki dezakegu.

5.1 Adibidea: Ardatz-talde baten definizioa

```
1 <group name="robot">
2         <chain base_link="base_link" tip_link="gripperBody" />
3 </group>
```

  Modu askotan definitu daitezkeen arren, metodorik errazena kate bat erabiltzea da. Honela, ardatz espezifiko batetik hasita, bukaerako ardatzerainoko ardatz zein lotura guztiak sartuko ditu talde berean. Adibidean, *base_link* ardatzetik *gripperBody* ardatzerainoko ardatz zein lotura guztiak robot taldea osatzen dutela definitzen da.

  Moveit!-en eragingailu baten definizioa agertzen da 5.2 adibidean.

5.2 Adibidea: efektore baten definizioa

```
1 <end_effector name="gripper" parent_link="gripperBody" group="robot"
```

  Honez gain robotaren poseak (robotaren lotura bakoitzerako posizio jakinak) ere defini daitezke, ardatz jakinen posizioak definituz (ikus: 5.3 adibidea).

5.3 Adibidea: Pose baten definizioa

```
<group_state name="rest" group="robot">
        <joint name="Joint0" value="0" />
        <joint name="Joint1" value="-0.2824" />
        <joint name="Joint2" value="1.7649" />
        <joint name="Joint3" value="-0.3177" />
</group_state>
```

Robotaren hainbat atalek instalazioko beste elementuekin edota bere buruarekin talka egin dezaketenez kolisio-matrize deritzona definitu behar da. Matrize honen bitartez zein ardatzek objektu edota robotaren beste ardatz batekin talka izan ahal dezaken definitzen da. 5.4 adibidean kolisio-matrize bat erazagutzeko zenbait adibide agertzen dira.

5.4 Adibidea: Kolisio-matrizearen erazagupena

```
<disable_collisions link1="geoJoint2" link2="gripperFinger2" reason="User" />
<disable_collisions link1="geoJoint3" link2="geoJoint4" reason="Adjacent" />
<disable_collisions link1="geoJoint3" link2="gripperBody" reason="Never" />
```

Talka bat gertatzeko hainbat arrazoi desberdin jar daitezkeen aren, erabilienak *User*, *Adjacent* eta *Never* dira. Adibidean, *geoJoint2* eta *gripperFinger2* ardatzen artean *User* arrazoia adierazi du erabiltzaileak laguntzaileak kolisio posibleak daudela detektatu ez duelako. Bi ardatz ondoan egoteaz gain beraien arteko kolisioa ezinezkoa bada laguntzaileak bi ardatz horien artean *Adjacent* arrazoiarengatik kolisiorik ez dela egongo adierazten du. Bi ardatzen arteko distantzia edota posizioarengatik beraien arteko kolisioa guztiz ezinezkoa bada, *Never* arrazoia erabiliko da.

- Loturen limiteen definizioa: *joint_limits.yaml*:
  Honako fitxategian robotaren URDF fitxategian aipatutako limiteak (posizio zein abiadurari dagozkienak) konfiguratzen dira Moveit!-ek kontuan hartzeko. Lotura baten konfigurazioa agertzen da 5.5 adibidean.

5.5 Adibidea: Lotura baten konfigurazioa

```
Joint0:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 30
    has_acceleration_limits: false
    max_acceleration: 0
```

Hemen *Joint0* loturaren limiteak espezifikatzen dira. Kasu honetan, lotura honek posizio zein abiadura limiteak dituela adierazten da.

- Alderantzizko zinematikaren konfigurazio fitxategia: *kinematics.yaml*:
  Fitxategi honetan mugimenduak planifikatzerakoan erabiliko den ebazlea, bere erresoluzioa eta ebazpen saiakera maximoak adierazten dira.

5.3 Moveit! sentsoreekin

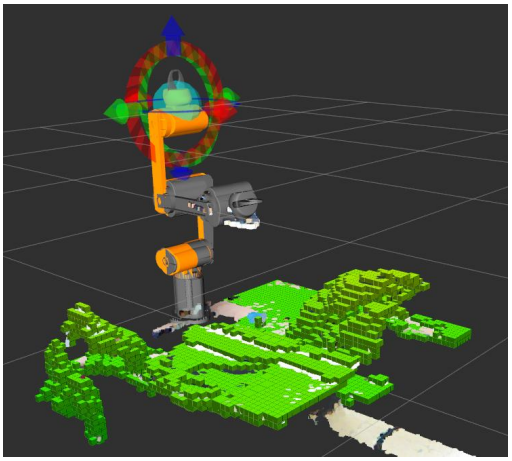
Moveit! gai da sentsoreen informazioa bereganatzeko eta informazio horrekin hainbat eginkizun betetzeko, hala nola, piezen manipulazioa egin edota oztopoak detektatu. Gure proiektuan Moveit! soilik oztopoak detektatzeko erabiliko da. Softwareak oztopoak ekidin ditzan konfigurazio fitxategi berri bat sortu behar da. Kasu honetan Kinect sentsorea erabili nahi genuenez, fitxategi honi *kinect.yaml* izena eman zaio. Bertan, sentsorea erabiltzeko plugin-a adierazten da lehenengoz. Honez gain Kinect-aren *pointcloud*-a zein *topic*-etan publikatzen den azaltzen da. Hau egin ondoren, sentsorearen informazioa zein distantzaraino hartu nahi dugun adierazi beharko da. Azkenik, *pointcloud* honen filtroa nola egin adierazi behar da, robotak berau oztopo bezala kontuan hartu ez dezan berau sentsorearen atzipen-eremuan agertzen bada, eta filtratutako *pointcloud* hau zein *topic*-etan publikatuko den adierazi beharko dugu.

5.4 Moveit! eta rViz

Simulazioak egiteko, Moveit! hainbat bistaratzaile zein simulatzailerekin elkarlanean ibili daiteke. Planifikatutako mugimenduak errepresentatzeko, Moveit!-ek kontroladore faltsuak (ingelesez, *fake controllers*) sortzen ditu robota bistaratzailean mugitzeko. Kontroladore hauek
fake_controllers.yaml konfigurazio fitxategiaren bidez definitzen dira.

Fitxategi honetan robotaren lotura bakoitzari izen bat eman behar zaio soilik (ez du zertan loturaren benetako izena izan behar).

Behin kontroladore horiek erazagututa, rViz bisualizadorean *MotionPlanning* izeneko elementua gehitzea besterik ez da behar. Pausu guztiak egin ondoren, 5.1 irudian azaldutakoa ikus dezakegu.



5.1 Irudia: *Moveit! rViz bisualizadorean*

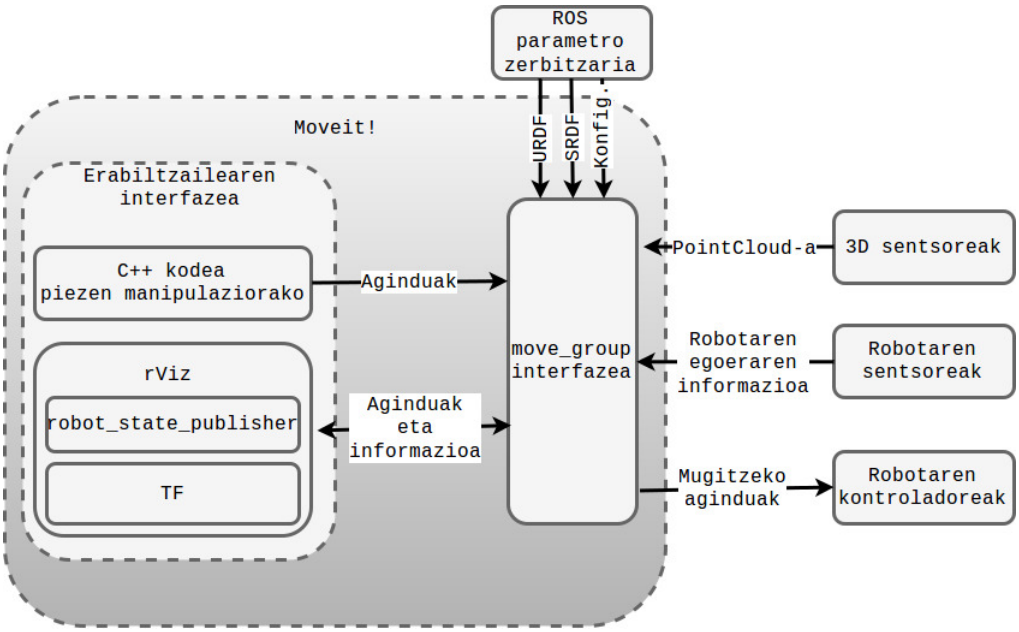
Irudian, laranja kolorez, robota mugitzean lortu nahi den posizioa azaltzen. Grisez, robotaren uneko posizioa adierazten da. Ingurunean kolore desberdinetan azaltzen diren "kubotxo"horiek Kinect sentsorearen informazioan oinarrituz softwareak detektatutako oztopoak adierazten dituzte. Ikus daitekeen bezala robota bera ez du oztopotzat hartzen, aurretik adierazitako filtroa aplikatzen baita.

5.5 Moveit! softwarearen eskema orokorra

Kapitulu honetan azaldutakoa agertzen zaigu 5.2 irudian laburbilduta.

Ikus dezakegun bezala, moveit! softwareak *move_group* izeneko interfazea darabil mugimenduen planifikazioaren arazoari aurre egiteko beharrezko guztia elkar konektatzeko (honi buruz gehiago arituko gara 9. kapituluan). Berauk hartzen du 3D sentsoreen (gure kasuan, oztopoak ekiditeko erabiliko dena) eta robotaren egoera jakinaraziko dion informazio guztia. Honez gain, ROSek darabilen parametro zerbitzaritik robotaren URDF eta SRDF fitxategiez gain, egindako moveit!-eko konfigurazioari buruzko informazioa lortuko du.

Informazio hori guztia edukita, erabiltzailearen interfazean agertzen zaigun C++ kodean agertzen diren aginduak exekutatu ahal izango dira. Kode hau ez da izango mugimenduak kalkulatzeko metodo bakarra, izan ere moveit!-ek, aurrerago azaldu dugun bezala, rViz bezalako bistaratzaileekin egin dezake lana, honen aldetik aginduak jasoz, eta bistaratzailean momentuko egoera ikusi ahal izateko informazioa luzatuz. Behin egin beharreko



5.2 Irudia: *Moveit! softwarearen eskema*

kalkulu guztiak eginda, interfaze honek aginduak emango dizkio robotaren kontroladoreei, hauek mugimendua deskriba dezaten.

5.6 Moveit! robot errealarekin

Simulazioaz gain, Moveit!-ek benetako robota kontrolatzea ahalbidetzen du, aginduak emanez. Horretarako beste konfigurazio fitxategi bat sortu beharko dugu. Fitxategi honen izena *controllers.yaml* da.

Fitxategi honetan robotak duen mugimendu-zerbitzaria zein den adierazi beharko da. Zerbitzari hauek (*follow_joint_trajectory* izenarekin ezagutzen direnak) robotaren konfigurazioan erazagutzen dira eta Moveit!-eko aginduak interpretatzeko balio dute. Behin zerbitzaria zein den adierazita, robotaren lotura bakoitzaren izenak azaldu behar ditugu, softwareak zerbitzariarekin komunikatzerakoan zein artikulazio mugitu behar duen adierazteko.

6. KAPITULUA

Instalazioa

6.1 Mover4 beso robotikoa

6.1.1 Sarrera

Mover4 CommonPlace Robotics etxeak garatutako beso robotikoa da. Beso robotiko hau RSAIT taldeak eskuratu zuen eta ROS framework-aren bitartez kontrolatuta erabili izan du taldeak frogak egiteko.

Robot hau lehenengo aldiz erabili da proiektu batean, aurretik soilik teleoperazio bidez kontrolatzeko frogak egin baitzituzten berarekin.

6.1.2 Egitura

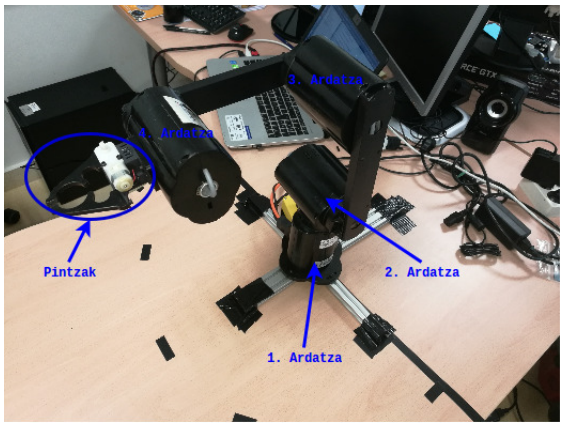
CommonPlace Robotics etxeko beso robotiko honek 4 ardatz eta pintza bat ditu. 4 ardatzak mugitzeko 4 serbomotore darabiltza eta pintzaren 2 behatzak ireki eta ixteko motore elektriko bat erabiltzen du. Besoaren 4 ardatzek 4 askatasun gradu eskaintzen dizkiote.

Besoaren 4 ardatzak, serbomotoreen bitartez darabiltzanez, badituzte bere posizio maximo eta minimoak, hauek 6.1 taulan azaltzen direlarik (ikus: 6.1 irudia ardatzen identifikaziorako).

Robot industriala ez denez, serbomotoreek eragin dezaketen indarra ere oso mugatua da. Mover4 beso robotikoak ezin ditu 500 gramo baino astunagoak diren objektuak mugitu.

Ardatza	Posizio minimoa (rad)	Posizio maximoa (rad)
1. ardatza	-2.58	2.58
2. ardatza	-0.47	0.47
3. ardatza	-0.68	0.68
4. ardatza	-2.26	2.26
Pintzak	0	2.58

6.1 Taula: Mover4 beso robotikoaren ardatzen posizio maximo eta minimoen taula



6.1 Irudia: *Mover4 beso robotikoaren ardatzak*

Honez gain, beso robotiko honen irismena oso mugatua da, 50cm-ko erradiodun zirkulo baten barneko posizioetara hel daiteke soilik.

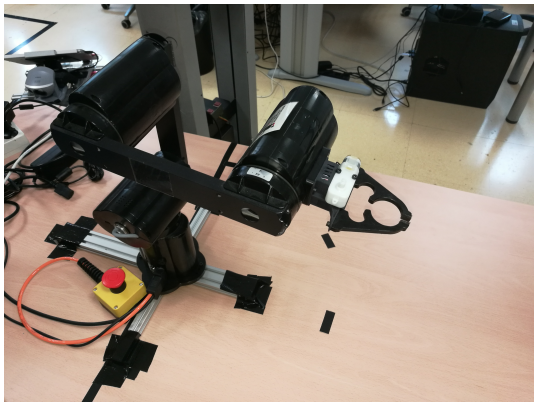
6.1.3 Kontrola eta komunikazioa

Robotarekin komunikatzeko Mover4 beso robotikoak CAN-USB interfazea darabil (ikus: 6.2 irudia). Interfaze hau erabiltzeko kontroladoreak instalatu beharko dira kontrol aginduak emango dituen ekipoan.

Larrialdiren baten kasuan, interfaze honekin batera larrialdietarako botoia dugu (ikus: 6.3 irudia). Botoia zanpatu eta biratuz gero komunikazioa eten eta robota gelditzen da.

Aginduak ematerakoan beso robotikoak bi aukera nagusi ematen ditu: bere software propioa erabiltzea ala ROS framework-arekin kontrolatzea. Gure proiektuan software propietarioa alde batera utziz ROSeko *cpr_mover* paketearen bidez kontrolatuko dugu beso robotikoa.

6.2 Irudia: *CAN-USB interfazea*



6.3 Irudia: *Mover4 beso robotikoa. Ezkerrean, larrialdietarako botoia*

6.2 Kinect sentsorea

6.2.1 Sarrera

Kinect-a Microsoft etxeak Xbox360 eta XboxOne kontsolentzat diseinaturiko 3D kamera bat da. Kamera honen bitartez inongo mandorik gabe ibil gaitezke bideojokoetan. Informazio gehiagorako Microsoft-en webgune ofizialera jo dezakegu [4].

Sentsore optiko hau bere ikusmen-angeluan (43º bertikalki, 57º horizontalki) dauden 1.228.800 puntuen XYZ posizioak eta koloreak hauteman ditzazke 30 aldiz segunduko. Gure proiektuan informazio hau guztia piezen eta oztopoen detekziorako erabiliko dugu.

6.4 Irudia: *Kinect sentsorea euskarrian*

6.2.2 Osagaiak

Kinect-ak informazio hori guztia lortzeko hainbat osagai biltzen ditu (ikus: 6.5 irudia):

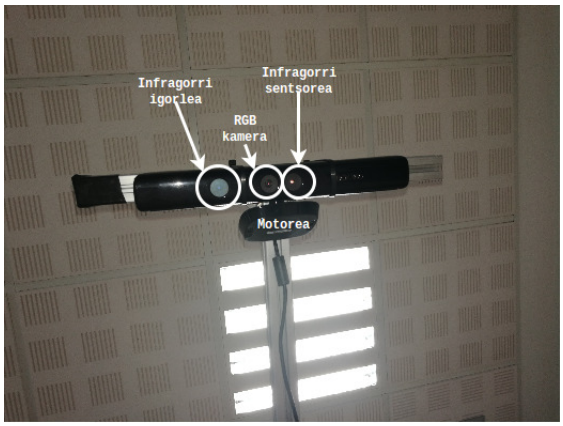
- Koloretako kamera:
  Kamera honen bitartez pointcloud-a osatzen duten puntu guztien RGB balioak lortu daitezke. Honen erresoluzioa 1280x960-koa da eta 30 FPStara dabil.

- Infragorri igorlea:
  Igorle honek izpi infragorriak igortzen ditu hauek objektuen aurka talka egin dezaten.

- Infragorri sentsorea:
  Talka egin duten izpien ailegatze-denborak kalkulatuz, objektuaren posizioa hautemateko balio du.

- Mikrofonoak:
  Proiektu honetan erabili ez arren, sentsoreak hainbat mikrofono ditu txertatuta mikrofono estereo bat balitz bezala.

- Motorea:
  Erabiliko ez den arren, sentsoreak bere oinarrian bertikalki pibotatzeko motore bat du. Honen bitartez objektu edota eremu handiagoak eskaneatu daitezke.



6.5 Irudia: *Kinect sentsorearen osagaiak*

6.2.3 Kontrola eta komunikazioa

Kinect-ak jasotzen duen informazioa erabiltzeko, USB baten bitartez pasatzen da informazioa konputagailura. USB hau 3.0 motakoa izan behar du, informazio asko denbora gutxian bidaltzen baitu sentsoreak, eta konputagailuak informazio hori denbora errealean jasotzeko gai izan behar du.

Kinect-a ROSen bidez kontrolatzeko beharrezkoa izango da berau kontrolatzeko kontrolagailuak eta freenect paketea instalatuta egotea.

6.3 Kinect sentsorearen euskarria

6.3.1 Sarrera

Proiektu honetan 3D kameraren funtzioa hain garrantzitsua izanik, robotaren ingurunea ahalik eta modu zehatzenean eta osatuenean detektatu nahi genuen. Hori dela eta Kinect sentsorea ingurunearen gainean, altuera batetara eskegitzea gure helburua lortzeko teknikarik onena zela aztertu genuen.

Kinect sentsorea posizio horretan jarri nahi izateak euskarri espezifiko baten beharra suposatu zuen. Hori dela eta, aluminioko profilak erabiliz, euskarri berezi bat diseinatu eta muntatu zen.

6.3.2 Arkitektura

Esan bezala, euskarria aluminiozko hiru profilez osatuta dago 6.6 irudian zenbaturik daudela. Euskarriaren osagaien neurriak honakoak dira, taula batean adierazita:

Elementua	Neurria (luzera x zabalera x altuera)(cm)
1. profila	100 x 3 x 3
2. profila	57 x 3 x 3
3. profila	43 x 3 x 3

6.2 Taula: Kinect sentsorearen euskarriaren neurrien taula

6.6 Irudia: *Kinect sentsorearen euskarria*

6.4 Manipulatu beharreko piezak

6.4.1 Sarrera

Mover4 beso robotikoak manipulatu beharreko piezak diseinatzerako orduan hainbat faktore hartu behar izan genituen kontuan:

- Robotaren pintza:
  Robotaren pintzaren forma kontuan hartu beharreko gauzarik garrantzitsuenen artean agertzen zaigu. Beso robotikoak pintzetan indar handirik ez duenez eta bi behatzez osaturik dagoenez manipulatu beharreko objektua arina eta simetrikoa izan behar zuela pentsatu zen.

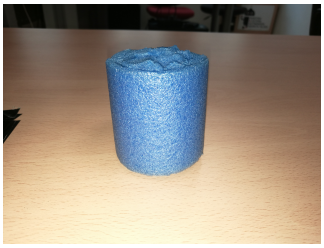
- Ingurunea:
  Ingurunea nolakoa da kontuan hartzea ere oso garrantzitsua da. Ingurunean destakatu beharreko objektu bat izan beharko du manipulatu beharreko piezak, ingurunetik desberdintzeko.

- Kinect-a:
  Kinect sentsorearen bereizmena ere kontutan hartu zen pieza diseinatzerakoan. Piezak forma geometriko simetrikoa balu errazago identifikatuko zukeen sentsoreak. Koloreari dagokionez, ingurunearen kolorearekin asko kontrastatzen zuen kolore bat hautatzea izango litzateke aukerarik onena.

Aipatutako elementu hauek kontuan hartuta, manipulatu beharreko pieza hori forma geometriko hutsa, simetrikoa, eta ingurunearekin kontrastatzen zuen kolore batekoa izan behar zuen.

6.4.2 Piezaren diseinua

Aurretik azaldutakoa kontuan hartuta, manipulatu beharreko piezaren diseinua egitera pasa ginen. Azkenean, 6cm-ko alturadun porexpanezko zilindro urdin argia diseinatu zen (ikus: 6.7 irudia).

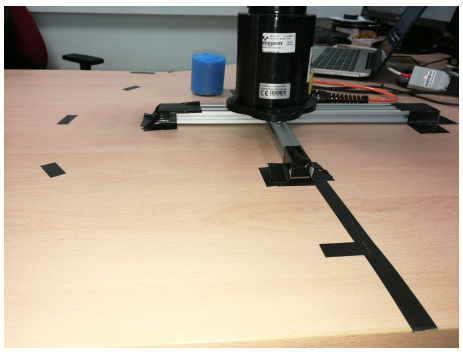


6.7 Irudia: *Manipulatu beharreko pieza*

6.5 Piezen posizioak

Piezen manipulazioa arinagoa egiteko, ingurunean 4 posizio desberdin markatu ziren. Manipulatu beharreko pieza zilindrikoa robotaren aurrealdean aurkitzen diren edozein 3 marketan (ikus: 6.8 irudia) jar daiteke robotak har dezan. Behin pieza hartuta, beso robotikoak 6.9 irudian agertzen den posizioan utziko du.

6.8 Irudia: *Piezak jartzeko posizioen markak*



6.9 Irudia: *Piezak uzteko posizioaren marka*

6.6 Oztopoak

Mover4 besoak bere irismena baino altuagoak ez diren edozein objektu erabili da oztopotzat: liburuak, boteilak...

6.7 Ingurune osoa

6.10 irudian kapitulu honetan azaldutako instalazioa agertzen da. Bertan, aurretik azaldutako elementu guztiak ikus ditzakegu: beso robotikoa, Kinect sentsorearen euskarria, Kinect sentsorea, manipulatu beharreko pieza, piezak jartzeko posizioak, hauek uzteko posizioa eta sistema guztia martxan jartzen duen PC-a.

6.10 Irudia: *Instalazio osoa muntatuta*

7. KAPITULUA

Sistema rViz bistaratzailean

7.1 Sarrera

Aurreko ataletan esan bezala, robotek modu seguruagoan eta eraginkorragoan funtziona dezaten derrigorrezkoak izaten dira simulazio zein bistaratzeak egitea software baten bidez, errealitatean probak egin aurretik.

Hori dela eta proiektu honetan instalazioan parte hartzen duten elementu guztien funtzionamendu egokia konprobatzeko eta errealitatean egin aurretik probak gauzatzeko rViz bistaratzailea erabili da. Bistaratzaile honen bitartez, hurrengo ataletan azalduko ditugun elementuen konportamendua ikusi eta aztertu ahal izan dugu. Honez gain, elementuak bistaratzailean ikusteko egin behar izandako konfigurazioak, software eta *plugin*-en instalazioa zein inplementatu beharreko kodea azalduko dira. Honako guztia lortzeko laguntza handikoak izan dira rViz-erako tutorialak, bere webgune ofizialen eskuragarri daudelarik [5].

7.2 Robotaren eredua

7.2.1 Sarrera

4. kapituluan azaldu genuen bezala, rViz-ek edozein objektu dinamiko ala estatiko bistaratzeko bere URDF eredua beharko du. Fitxategi honen bitartez objektuaren ezaugarriak

adieraziko ditugu.

Robotaren eredua bistaratzeaz gain beso robotikoaren kontrola gauzatzeko, gainera, rVizerako plugin bat beharko du. Plugin honek robotaren modeloa bistaratzailean agertzeaz gain, bere kontrola edukitzen ahalbidetuko du (kontrolari buruz 7.3 atalean jardungo dugu).

7.2.2 URDF fitxategia

rViz bistaratzailean ikusi nahi dugun edozein elementu bezala, Mover4 beso robotikoak ere beharko du URDF fitxategi bat berau definitzeko. Beso robotiko honen definizioa *CPRMover4.urdf.xacro* fitxategian aurkitu dezakegu. Fitxategi hau sortu eta modifikatzerakoan 4. kapituluan azaldutako adibideetatik abiatu gara.

Fitxategi honetan, hasteko, beso robotikoaren elementu guztien definizioa egin da (ikus E1 eranskina).

4.1 Adibidea jarraituz sortutako fitxategi honetan robotaren elementu guztiak (5 loturak, pintzaren euskarria eta pintzaren bi behatzak) erazagutu dira <*link*> etiketa erabiliz.

Beraien izenak definitzeaz gain, bistaratzailean elementu bakoitzak edukiko duen posizioa, materiala, kolorea, eta ehundurak adierazteko *.obj* fitxategiak definitu dira.

Honez gain, robotaren elementuen arteko loturak ere definitu behar izan dira <*joint*> etiketaren bitartez, E2 eranskinean agertzen den bezala.

Bertan ikus dezakegun bezala, robotaren elementuak mugitzen dituzten serboak eta pintza ireki eta izten duen motorea erazagutu dira <*joint*> etiketa erabiliz.

Aurreko kasuan bezala, serbo hauen izena eta bistaratzailean okupatuko duten posizioa adierazi dira. Honez gain, serbo hauek dituzten abiadura, posizio zein indar mugak adierazi dira muga hauek guztiak ekoizleak gomendaturikoak izanik. Azkenik, serboak definitu direnez, loturaren mota *revolute* motakoa dela adierazi da.

7.2.3 *cpr_mover4.launch* fitxategia

Aurreko atalean diseinatutako eredua bistaratzailean ikusteko ezinbesteko da berau *.launch* fitxategi baten bitartez martxan jartzea.

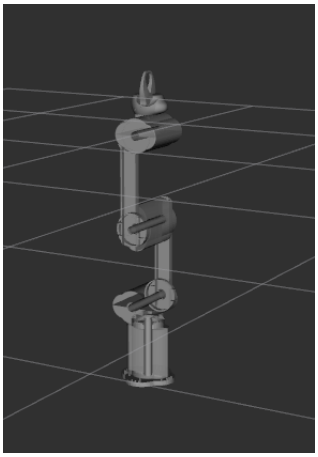
Hori lortzeko hurrengo *.launch* fitxategia sortu zen (7.1 adibideko kodean). Bertan, bistaratu beharreko ereduaren URDF fitxategia adierazteaz gain, *robot_state_publisher* deritzon argitaratzailea martxan jartzen dugu. Argitaratzaile honek uneoro robotaren egoera adieraziko dio rViz bistaratzaileari modeloaren posea uneoro dagokiona bistaratzeko. Azkenik, rViz bistaratzailea jarriko da martxan.

7.1 Adibidea: *cpr_mover4.launch* fitxategia

```
<launch>
    <param name="robot_type" value="mover4"/>
    <param name="robot_description" textfile="$(find cpr_rviz_plugin)/CPRMover4.urdf.xacro" />

    <param name="use_gui" value="True"/>

    <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher">

    </node>

    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find cpr_rviz_plugin)/cpr_mover4.rviz" />
</launch>
```

7.2.4 Ereduaren bistaratzea rViz-en

Behin aurreko ataletan definitutako URDF fitxategia eta *.launch* fitxategia kodetuta, berau exekutatu dezakegu. *cpr_mover4.launch* fitxategia exekutatuz gero robotaren eredua erakusten duen rViz-eko mundu bat ikusiko dugu, 7.1 irudian agertzen dena bezalakoa.



7.1 Irudia: *Mover4 beso robotikoaren eredua rViz-en*

7.3 Robotaren kontrola eta mugimenduen bistaratzea

7.3.1 Sarrera

Robotaren eredua bistaratzearekin ez da nahikoa elementu guztiek ondo egingo al duten aztertzeko edota eredua berau ondo definituta dagoenentz ziurtatzeko. Hori dela eta, bistaratzailean robotari hainbat mugimendu deskribatzeko eskatuko diogu, bistaratzailean agertzen den modeloa modu egokian mugitzen dela ziurtatzeko. Honakoa oso interesgarria eta beharrezkoa izango da benetako beso robotikoarekin lan egin aurretik.

7.3.2 Robotaren kontroladorea

6. kapituluan azaldu genuen Mover4 beso robotikoaren morfologia eta kontrola. Kontrol hori gauzatzeko CAN/USB interfazea erabiltzen zela ere azaldu genuen.

Beso robotikoa errealitatean zein rViz bistaratzailean kontrolatzeko, berarentzako kontrol aginduak bidaltzeko gai den kontroladore bat beharko dugu. Kontroladore hau *cpr_mover* ROSeko paketean aurkituko dugu.

Pakete honetan rViz-en bistaratzeko zein errealitatean CAN-USB interfazearen bitartez aginduak emateko kontroladoreak ekartzen ditu. Honez gain, hurrengo kapituluetan azalduko dugun Moveit! softwarea erabiltzeko hainbat zerbitzu jartzen ditu martxan.

Beraz, beso robotikoarekin edozein mugimendu deskribatzeko (errealitate zein bistaratzailean), lehenik eta behin Mover4 beso robotikoaren nodoa (exekutagarria) jarri beharko dugu martxan honako komandoaren bidez:

```
rosrun cpr_mover cpr_mover
```

Exekutagarri honen bitartez rViz-ekin komunikatzeko beharrezko prozesuak martxan jartzeaz gain, ematen dizkiogun aginduak (teleoperazio bidez) jasotzeko prest jarriko da.

Hurrengo bi ataletan rViz-en probak egiteko erabili ziren bi kontrol mota azalduko dira: programa bidezkoa eta Mover4 besoarentzako rViz-erako *plugin*-arekin egindakoa.

7.3.3 Programa bidezko teleoperazioa

Robotaren kontrola gauzatzeko lehenengo pausua besoaren ardatzak guk definitutako posizio batetara mugitzeko C++ programazio lengoaian inplementatutako programa bat sortzea izan zen.

Programa honek *joint_states topic*-a irakurtzen robotaren uneko posizioa ezagutzen du. Posizio hori ez badator bat guk emandakoarekin, ardatzei mugitzeko agindua bidaltzen die *CPRMoverJointVel topic*-ean idatziz.

Besoak mugitu behar duen posizioa programari jakinarazteko, espreski diseinatutako */go_to_pos topic*-ean idatzi beharko dugu */go_to_pos* mezu-motako mezu bat. Mezu hau 6 zenbakiz osatuta dago. Zenbaki bakoitza robotaren 6 artikulazioen posizioei (radianetan) dagokie.

Robota nahi dugun posizioetara mugitzeko, ROSek eskaintzen digun *rostopic pub* komandoa erabiltzea besterik ez dugu behar. Komando honek edozein *topic*-etan mezuak idaztea ahalbidetzen digu, beratara harpidetutako nodoek irakur dezaten. Beraz, besoa mugitzeko bi pausu hauek eman beharko ditugu:

- *teleop_proba.cpp* kodea exekutatu:
  Lehenik eta behin besoa mugitzeko programa martxan jarri beharko dugu honakoa exekutatuz:

  ```
  rosrun teleop_proba teleop_proba
  ```

  Hau egin ostean dagoeneko */go_to_pos topic*-aren bidez aginduak jasotzeko prest egongo gara.

- Mugitu nahi dugun posizioak publikatu:
  Aurretik aipatutako *rostopic pub* komandoa erabiliz, besoaren ardatz bakoitza zein posizioetara mugitu nahi dugun adierazi beharko dugu:

  ```
  rostopic pub /go_to_pos 0 0 0 0 0 0
  ```

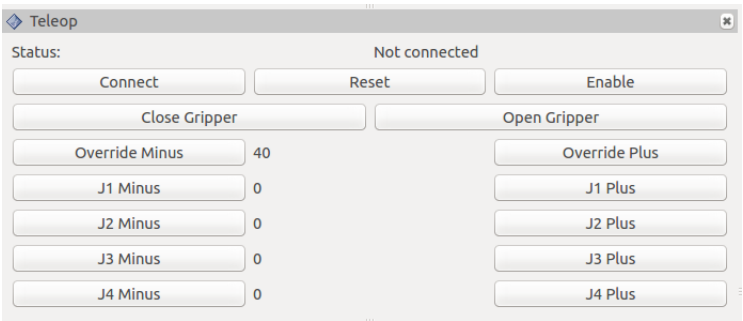
  Aurreko komando hau exekutatzen badugu, robotaren ardatz guztiak 0 radianetako posizioan jarriko dira.

7.3.4 Teleoperazioa rViz-etik

Beso robotikoa programa bidez mugitu ondoren, Mover4 robotarentzat egindako rViz-erako *plugin*-a erabiliz kontrolatu genuen bere mugimendua. *Plugin* honek rViz bistaratzailean robotaren artikulazioak mugitzeko kontrol panel bat gehitzen zuen. Panel honen bitartez, programan egiten genuen bezala, *joint_states topic*-etik irakurtzen zuen uneko posizioa eta *CPRMoverJointVel topic*-ean idazten zuen agindutako posiziora mugitzeko. 7.2 irudian kontrol panel hori ikus dezakegu.

Panel hau bistaratzeko honako *.launch* fitxategia exekutatu beharko dugu:

```
roslaunch cpr_rviz_plugin cpr_mover4.launch
```



7.2 Irudia: *Mover4 beso robotikoaren teleoperaziorako kontrol panela*

Robota mugitzeko, hasteko, *Connect* botoia sakatu beharko dugu, modu honetan rViz *joint_states topic*-a irakurtzeko prest jarriko da. Honen ondoren *Reset* botoia sakatzen badugu rViz-ek robotaren uneko posizioa irakurriko du *joint_states topic*-etik eta uneko posizioa bistaratuko du. Azkenik, *Enable*-ri sakatuz, benetako robotarekin konektatuko da (hau aurrerago egingo dugu).

Pausu horiek guztiak eginik, dagoeneko robotaren artikulazioak oso modu intuitiboan mugitu ahal izango ditugu *Jx Minus* eta *Jx Plus* botoiekin (x artikulazioaren zenbakia izanik). Honez gain, pintzak ireki eta itxi ahal izango ditugu *Open Gripper* eta *Close Gripper* botoiekin.

7.4 Ingurunea rViz-en

7.4.1 Sarrera

Beso robotikoaz gain, rViz-en robotaren ingurune osoa ere diseinatu eta bistaratu behar izan zen. Inguruneko elementuak bistaratuz, rViz-en benetako ingurunearen errepresentazio zehatzagoa eta hobeagoa lortuko dugu eta, batez ere, errealitatean gerta daitekeena askoz modu zehatzagoan aurreikusi dezakegu.

Hori dela eta, ingurunean agertzen den beste objektua diseinatu dugu: Kinect sentsorearen euskarria.

7.4.2 URDF fitxategia

Mover4 beso robotikoarekin gertatzen zen bezala, 3D eredu honek ere URDF fitxategi bat beharko du rViz-en bistaratzeko. URDF fitxategi hau egiteko, *urdf_tutorials* paketea eta 4. kapituluko objektu estatikoen diseinurako adibideetan oinarritu gara.

Euskarri hau 3 aluminiozko profilez osatuta dago. URDF fitxategian 3 profil hauek zehazteaz gain, beraien arteko loturak ere diseinatu dira. 7.2 kodean profilen materiala definitu dugu. 7.3 adibidean, aldiz, euskarriaren hiru ardatzak eta beraien posizioak adierazi ditugu. Azkenik, 7.4 adibidean hiru ardatzen arteko loturak diseinatu ditugu.

7.2 Adibidea: Profilen materialaren definizioa

```
<material name="grey">
    <color rgba=".60 .60 .60 1"/>
  </material>
```

7.3 Adibidea: Euskarriaren 3 ardatzen definizioa

```
  <link name="camera_base">
    <visual>
      <geometry>
        <box size="1 .03 .03"/>
      </geometry>
      <origin rpy="1 1.57075 0" xyz="0 0 0.5"/>
      <material name="grey"/>
    </visual>
  </link>

```

```
11  <link name="horizontal">
12    <visual>
13      <geometry>
14        <box size=".57 .03 .03"/>
15      </geometry>
16      <origin rpy="0 0 0" xyz=".16 0 1"/>
17      <material name="grey"/>
18    </visual>
19  </link>
20
21  <link name="final">
22    <visual>
23      <geometry>
24        <box size=".43 .03 .03"/>
25      </geometry>
26      <origin rpy="0 0 1.57075" xyz="0 0 1"/>
27      <material name="grey"/>
28    </visual>
29  </link>
```

7.4 Adibidea: Euskarriaren ardatzen arteko loturen definizioa

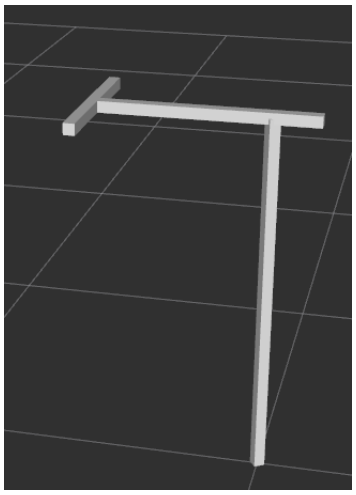
```
1  <joint name="base_to_horizontal" type="fixed">
2     <parent link="camera_base"/>
3     <child link="horizontal"/>
4     <origin xyz="0 0 0"/>
5   </joint>
6
7   <joint name="horizontal_to_final" type="fixed">
8     <parent link="horizontal"/>
9     <child link="final"/>
10    <origin xyz=".46 0 0"/>
11  </joint>
```

7.4.3 *display.launch* fitxategia

Aurreko kasuan bezala, rViz bistaratzailean diseinatutako eredu hau bistaratzeko *.launch* fitxategi bat beharko dugu. Fitxategi honetan, beso robotikoaren kasuan bezala, *robot_state_publisher* bat sortuko dugu, robotaren (kasu honetan, euskarriaren) egoera bistaratzeko. Euskarria rViz-en azaltzeko honakoa exekutatu beharko dugu:

```
roslaunch urdf_tutorial display.launch model:='03-origins.urdf'
```

Komando honen bidez *display.launch* fitxategia exekutatuko dugu *03-origins.urdf* fitxategia (euskarriarena) kargatuz. 7.3 irudian rViz-en euskarriaren eredua ikus dezakegu.



7.3 Irudia: *Kinect sentsorearen euskarria rViz-en*

7.5 Ikusmen artifiziala: *freenect* paketea

7.5.1 Sarrera

4. kapituluan komentatu genuen bezala, gure proiektuan oso garrantzitsua izango da ikusmen artifiziala. Gure proiektu honetan berau garatzeko Kinect sentsorea erabiliko dugu. 3D kamera honek bere ingurua eskaneatzen du *pointcloud* edo puntu-laino bat sortuz. *Pointcloud* honek inguruko puntu bakoitzaren posizio eta kolorearen informazioa lortzen du.

Hurrengo ataletan Kinect sentsorearen informazioa rViz bistaratzailean ikusteko eginiko pausuak azalduko dira.

7.5.2 *freenect* paketea

Kinect kamera erabiltzen hasteko lehenengo pausua berarekin lan egiteko kontroladoreak eta *plugin*-ak dakartzan pakete bat instalatzean datza. Hainbat aukera egon arren, *freenect* paketea instalatu genuen, ezagunetako bat baita.

Pakete honek Kinect sentsorearen informazioa hainbat *topic* desberdinen bidez argitaratzen du. Argitaratzen dituen topic guztietatik bi dira gure proiektuan erabiliko direnak: */camera_depth_registered_points* eta */camera_rgb_image_raw*. Lehenengo *topic*-ean sentsoreak erregistratutako puntu bakoitzaren xyz posizioa eta rgb kolore balioa argitaratzen

ditu, *pointcloud*-a. Bigarrengo *topic*-ean, aldiz, rgb kameraren informazioa argitaratzen da (*topic* hau Moveit! aztertzen dugunean erabiliko dugu).

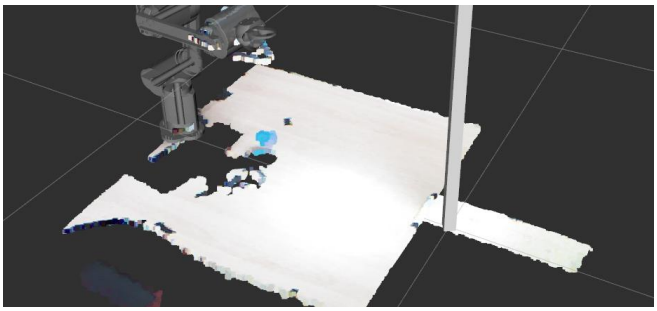
Paketea instalatu ondoren erabiltzen hasteko behin Kinect-a ordenagailura konektatuta hurrengo komandoa exekutatu behar da:

```
roslaunch freenect_launch freenect.launch
```

.launch fitxategi honek *freenect* paketeko programa batzuk jartzen ditu martxan. Horietako batek konektatuta dagoen Kinect sentsorea identifikatzen du, beste batek aginduak ematen dizkio eta azkeneko batek informazioa *topic* desberdinen bitartez argitaratzen du.

7.5.3 Kinect-aren *pointcloud*-a rViz-en

freenect paketeko *.launch* fitxategia exekutatu ondoren dagoeneko Kinect sentsorearen informazioa bistaratu dezakegu rViz-en. Hori lortzeko, rViz bistaratzailean *pointcloud2* motako elementu bat gehitu beharko dugu. Modu honetan, *pointcloud*-aren puntu guztien xyz eta rgb balioak ikus ditzakegu. Behin elementu hori gehituta, datuak */camera_depth_registered_points topic*-etik hartzeko adieraziko diogu. Modu honetan, 7.4 irudian ikusten den bezala, Kinect-ak sortzen duen *pointcloud*-a ikus dezakegu rViz-en.



7.4 Irudia: *Kinect sentsorearen* pointcloud-*a rViz-en*

7.6 Posizio transformatuak: *tf* paketea

Bistaratzailean agertzen diren elementu guztiak errealitatean agertzen diren posizioetan agertu daitezen beraien arteko erlazioa argitaratzea beharrezkoa da. Horretarako 3. kapituluan aipatutako *tf* paketea erabiliko dugu. Pakete honen bitartez rViz bistaratzaileari instalazioko objektu guztiak non dauden eta beraien arteko eralzioak azalduko dizkiogu.

tf paketean oinarrituta, gure proiektuan aplikatzeko *tf_proba* paketea sortu genuen. Bertan, posizio transformatuak argitaratzeko C++ kodea eta berau rViz-en aplikatzeko *.launch* fitategia sortu genuen.

7.5 adibideko kodean posizio-transformatuak argitaratzeko erabilitako *tf_publisher_proba.cpp* kodea dugu. Bertan, instalazioko elementu guztiak kokatzerakoan, erlazio guztiak kinectaren euskarriarekiko egingo direla adierazten da. Modu honetan, sistemako edozein elementuren posizioa adierazterakoan, euskarriaren erreferentzia-sistemarekikoa izango da.

7.5 Adibidea: tf_publisher_proba.cpp programa

```
 #include <iostream>
using namespace std;
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv){

  ros::init(argc, argv, "tf_publisher_proba");

  ros::NodeHandle nh;
  ros::NodeHandle nh_priv("~");
  string frame_id, child_frame_id;
  double x, y, z, yaw, pitch, roll;


  nh_priv.param<std::string>("frame_id", frame_id, "camera_base");
  nh_priv.param<std::string>("child_frame_id", child_frame_id, "base_link");

  nh_priv.param<double>("x", x, 0.0);
  nh_priv.param<double>("y", y, 0.0);
  nh_priv.param<double>("z", z, 0.0);

  nh_priv.param<double>("yaw", yaw, 0.0);
  nh_priv.param<double>("pitch", pitch, 0.0);
  nh_priv.param<double>("roll", roll, 0.0);

  ROS_INFO("frame_id:%s child_frame_id:%s",frame_id.c_str(),child_frame_id.c_str());
  ROS_INFO("x:%f y:%f z:%f yaw:%f pitch:%f roll:%f",x,y,z,yaw,pitch,roll);
```

```
29
30    tf::Quaternion qua;
31
32    static tf::TransformBroadcaster br;
33    tf::Transform transform;
34    transform.setOrigin( tf::Vector3(x, y, z) );
35    qua.setEuler(yaw, pitch, roll);
36    transform.setRotation(qua);
37
38   ros::Rate loop_rate(10);
39
40    while (nh.ok())
41          {
42              ros::spinOnce();
43              br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), frame_id,
      child_frame_id));
44              loop_rate.sleep();
45          }
46
47    return 0;
48  };
```

Honekin batera, *tf_publisher_proba.cpp* kodea exekutatzeko eta elementuen arteko erlazioa adierazteko *tf_publisher_proba.launch* fitxategia sortu da (ikus 7.6 adibideko kodea). Fitxategi honetan instalazioko elementu bakoitzaren posizio-erlazioa adierazten da Kinect sentsorearen euskarriarekiko.

7.6 Adibidea: tf_publisher_proba.launch fitxategia

```
1   <launch>
2
3       <node name="tf_publisher_camera_base" pkg="tf_proba" type="tf_publisher_proba" output="screen
        ">
4           <param name="frame_id" type="string" value="camera_base"/>
5           <param name="child_frame_id" type="string" value="base_link"/>
6
7           <param name="x"  value="0.83"/>
8           <param name="y"  value="0.0"/>
9           <param name="z"  value="0.0"/>
10
11          <param name="yaw"  value="0.0"/>
12          <param name="pitch"  value="0.0"/>
13          <param name="roll"  value="3.14159"/>
14      </node>
15
16      <node name="tf_publisher_final_camera" pkg="tf_proba" type="tf_publisher_proba" output="
        screen">
17          <param name="frame_id" type="string" value="final"/>
18          <param name="child_frame_id" type="string" value="camera_link"/>
```

```
19
20          <param name="x"  value="-0.02"/>
21          <param name="y"  value="0.05"/>
22          <param name="z"  value="0.96"/>
23
24          <param name="yaw"  value="1.5708"/>
25          <param name="pitch"  value="0.0"/>
26          <param name="roll"  value="0.0"/>
27      </node>
28  </launch>
```

7.6 adibideko kodean ikus dezakegunez, aurretik garatutako *tf_publisher_proba.cpp* koderen bi exekuzio egiten dira aldiberean. Lehenengo exekuzioan, Kinect-aren euskarriaren oinarriaren eta Mover4 beso robotikoaren arteko posizio-erlazioa adierazten da xyz eta roll,pitch eta yaw parametroen bidez (posizio eta errotazioaren parametroak). Bigarrengoan, aldiz, Kinect sentsorea zintzilik dagoen aluminiozko profiletik (*final* ardatza) euskarriaren oinarriarekiko posizio eta errotazio argumentuak finkatzen dira.

Modu honetan, instalazioko elementu nagusienak (Kinect sentsorea, euskarria eta beso robotikoa) erreferentzia-sistema amankomun batean egongo dira definituta, eta beraien arteko posizio-transformatuak egin ahal izango dira.

7.7 Ingurune osoa rViz-en

Instalazioko elementu guztiak eta beraien arteko posizio-erlazioak definituta, dagoeneko sistema osoa bistaratu dezakegu rViz bistaratzailean. Horretarako, instalazioan parte hartzen duten nodo eta programa guztiak exekutatu beharko dira hurrengo ordenean:

- *cpr_mover* programa:
  rViz bistaratzailean beso robotikoaren mugimenduak ikusteko beharrezkoa da programa hau exekutatzea. Hori egiteko, honakoa exekutatu beharko dugu terminalean:

  ```
  rosrun cpr_mover cpr_mover
  ```

- *freenect.launch* fitxategia:
  Behin Kinect sentsorea konputagailura konektatuta, nodo hau abiatzean 3D kameraren informazioa jasoko dugu:

  ```
  roslaunch freenect_launch freenect.launch
  ```

- Euskarriaren URDF eredua bistaratu:
  Euskarria rViz bistaratzailean ikusteko *urdf_tutorial* paketean garatutako URDF eredua *display.launch* fitxategiaren bitartez bistaratuko dugu:

  ```
  roslaunch urdf_tutorial display.launch model:='03-origins.urdf'
  ```

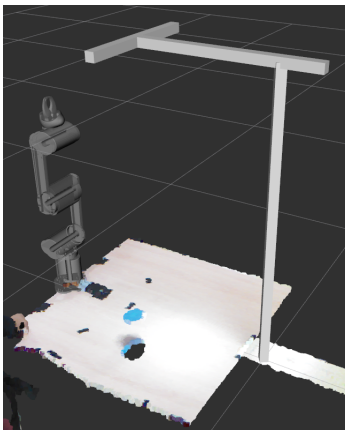
- Posizio transformatuak:
  rViz abiatu aurretik instalazioko elementuen arteko erlazioak argitaratu beharko ditugu. Hori lortzeko honakoa exekutatuko dugu:

  ```
  roslaunch tf_proba tf_publisher_proba.launch
  ```

- *cpr_mover4.launch* fitxategia:
  AZkenik, fitxategi honen bitartez rViz bistaratzailea abiatuko dugu eta instalazio osoa ikusi ahal izango dugu. Horretarako honakoa exekutatuko dugu:

  ```
  roslaunch cpr_rviz_plugin cpr_mover4.launch
  ```

Aurretik adierazitako guztia exekutatu odoren, 7.5 irudian azaldutakoa ikusiko dugu rViz bistaratzailean.



7.5 Irudia: *Ingurune osoa rViz bistaratzailean*

8. KAPITULUA

Alderantzizko zinematika Moveit! softwarearekin

8.1 Sarrera

Mover4 beso robotikoak bere helburua lortzeko ezinbestekoa izan da mugimendu-planeatzaile baten laguntza izatea. Mugimendu planeatzaileak robota nahi dugun posizioetara modu eraginkorrean eta errazean mugitzea ahalbidetzen digu. Robot baten mugimenduak gauzatzeko, 5. kapituluan esan bezala, bi teknika nagusi aplika daitezke: zinematika hutsa edota alderantzizko zinematika. Zinematika hutsa aplikatzea oso lan neketsua bihur daiteke askatasun gradu askoko robotekin lan egiten badugu. Kasu honetan robotaren ardatz bakoitzaren posizioak banaka-banaka definitu beharko genituzke mugimendu bat gauzatzeko. Alderantzizko zinematikarekin, aldiz, nahikoa da bukaera posizio bat finkatzea mugimendua sor dadin. Kalkuluak askoz ere konplexuagoak izan arren, gaur egun Moveit! bezalako softwareek modu azkar eta eraginkorrean egiten dituzte kalkulu horiek eta bukaerako posizio batera mugitzeko tarteko pausuak kalkulatzeko gai dira.

Gure proiektuan, esan bezala, Moveit! mugimendu-planeatzailea erabiliko dugu proiektu honen bi helburu nagusiak betetzeko: piezak manipulatzeko eta mugimenduak modu seguru batean (oztopoak ekidinez) gauzatzeko.

Hurrengo ataletan Moveit!-en konfigurazio osoa azalduko dugu: hasierako konfigurazio fitxategiak, sentsoreen atzeraelikadura lortzeko eta azkenik, rViz bistaratzailearekin batera erabiltzeko egindako konfigurazioa. Softwarea erabiltzeko tutorialak bere webgune ofizialean topa ditzakegu [6].

8.2 Moveit!-en laguntzailea

8.2.1 Sarrera

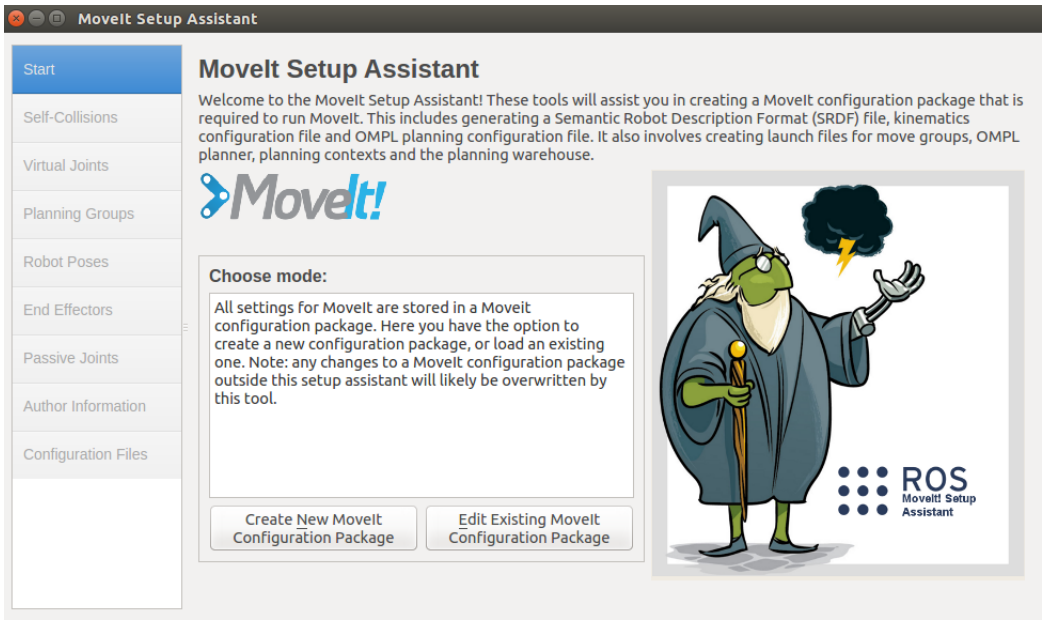
Moveit! softwarea hainbat konfigurazio-fitxategiren bitartez konfiguratzen da. Konfigurazio fitxategi guztiak zerotik sortzea oso lan neketsua bilakatu daitekeenez, 5. kapituluan adierazi genuen bezala, laguntzaile bat dakar berarekin. Laguntzaile honen bitartez oinarrizko konfigurazio fitxategiak sortuko ditugu.

8.2.2 Moveit!-en laguntzailearen exekuzioa

Moveit! softwarearen laguntzailea abiatzeko, honakoa exekutatu beharko dugu terminalean:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

Komandoa exekutatu ostean, laguntzailearen interfazea azalduko zaigu (ikus: 8.1 irudia).



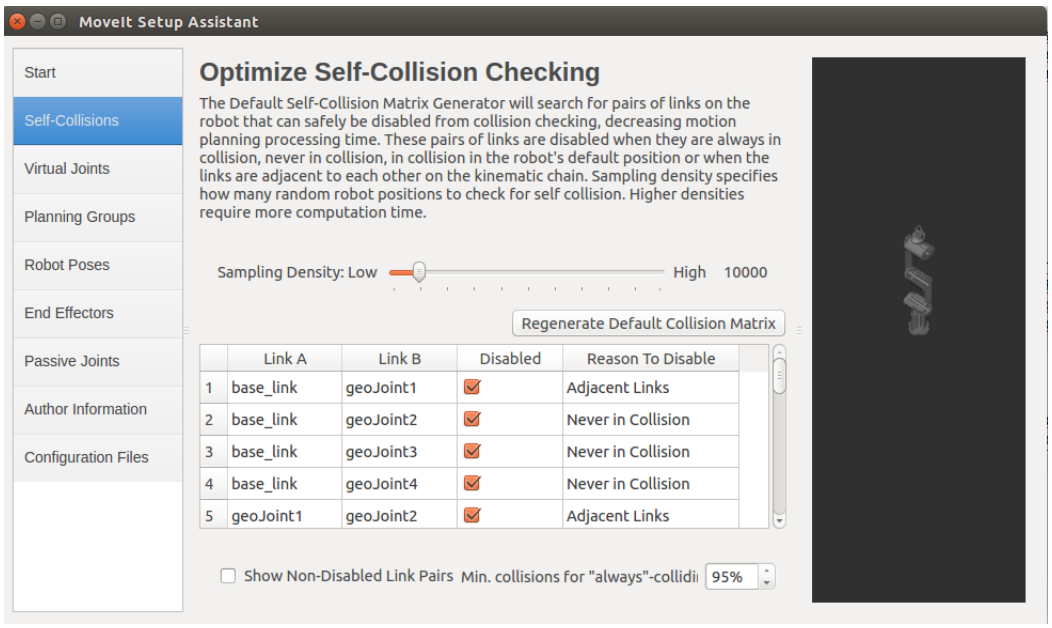
8.1 Irudia: *Moveit!-eko laguntzailearen interfazea*

Konfigurazio pakete berri bat sortu nahi badugu *Create New Moveit Configuration Package* aukeratuko dugu. Bertan sakatuz gero mugitu nahi dugun robotaren URDF fitxategia zein den galdetuko digu. Gure kasuan URDF fitxategi hau *cpr_rviz_plugin* paketeko

CPRMover4.urdf.xacro fitxategia izango da. Behin hori adierazita, laguntzaileak fitxategia kargatuko du eta hurrengo konfigurazioak egiten utziko digu.

Behin fitxategia kargatuta, hurrengo pausora joko dugu, kolisio-matrizea sortzera. Matrize honen bitartez ardatzen arteko kolisioak aztertuko ditugu hauek ekiditeko. Kolisio-matrize hau laguntzaileak gure nahien arabera sortuko duen arren, berauk detektatutako kolisio posibleak bakarrik hartuko ditu kontutan, geroago berau definitzen den fitxategian aldaketak egin behar izango ditugu.

Gure kasuan askatasun gradu gutxiko beso robotikoa denez, ez dugu matrize oso konplexua sortuko, nahikoa izango da kolisio-detekzio txikia gauzatzea. Horretarako, *Sampling density* parametroa *low*-n utziko dugu, eta *Regenerate Default Collision Matrix* aukerari emango diogu. Horrela, kolisio-matrizea sortuko dugu (ikus: 8.2 irudia).



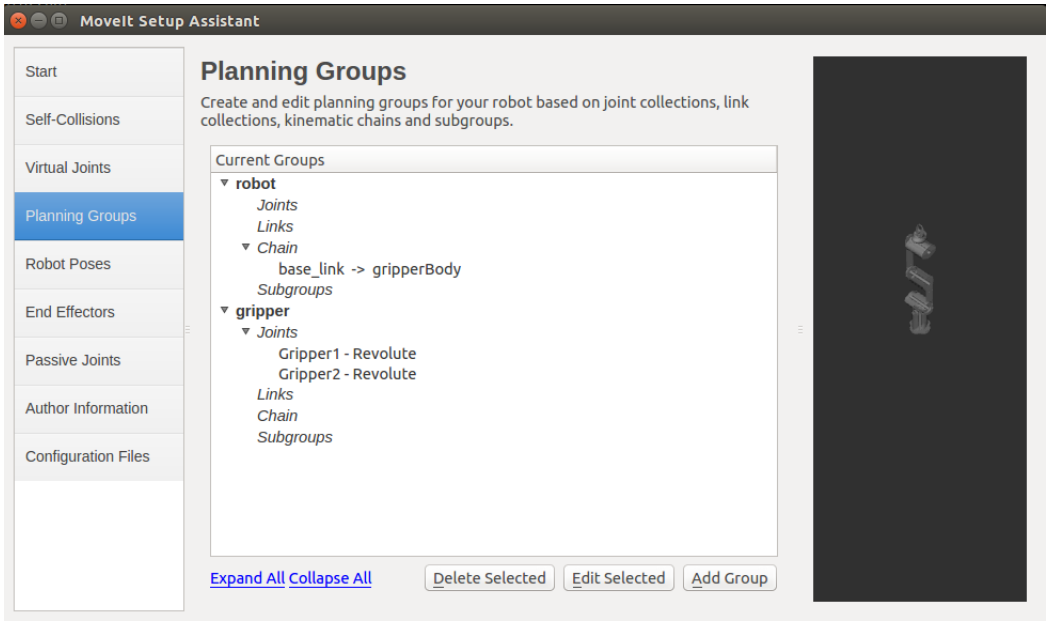
8.2 Irudia: *Kolisio-matrizearen sorkuntza*

Laguntzaileak eskatzen duen hurrengo pausoa, lotura birtualak sortzearena (*Virtual Joints*) alde batera utziko dugu, ez baita beharrezkoa erreferentzi-sistema berri bat sortzea Moveit!-ek lan egin dezan, *tf* fitxategia dagoeneko konfiguratuta baitugu.

Hurrengo pausoa oso garrantzitsua da. Bertan, mugimenduak planeatu nahi ditugun ardatz-taldeak definituko ditugu. Gure kasuan, pintzak alde batera utziz (hori beste modu batetara definitu nahi dugulako), robotaren ardatz guztiak *robot* izeneko ardatz-taldean sartuko dugu. Honen arrazoia sinplea da: mugimenduak robotaren ardatzentzako kalkulatu nahi ditugu, pintzek ez baitute zer eginik robota mugitzen ari den bitartean, beraien lana pieza

hartu eta uztearena baita. Pintzak aparteko talde batean sartuko ditugu, eta mugimenduak ez ditugu planeatuko beraientzat.

Hori dela eta, 8.3 irudian ikus dezakegun bezala, *robot* ardatz-taldea robotaren oinarritik pintzaren oinarriraino doazen ardatz guztiek osatzen dute, eta kate (*chain*) baten bitartez definitu dugu, modu errazagoan adierazteko. *Gripper* taldea, aldiz, pintzaren bi behatzek osatzen dute.

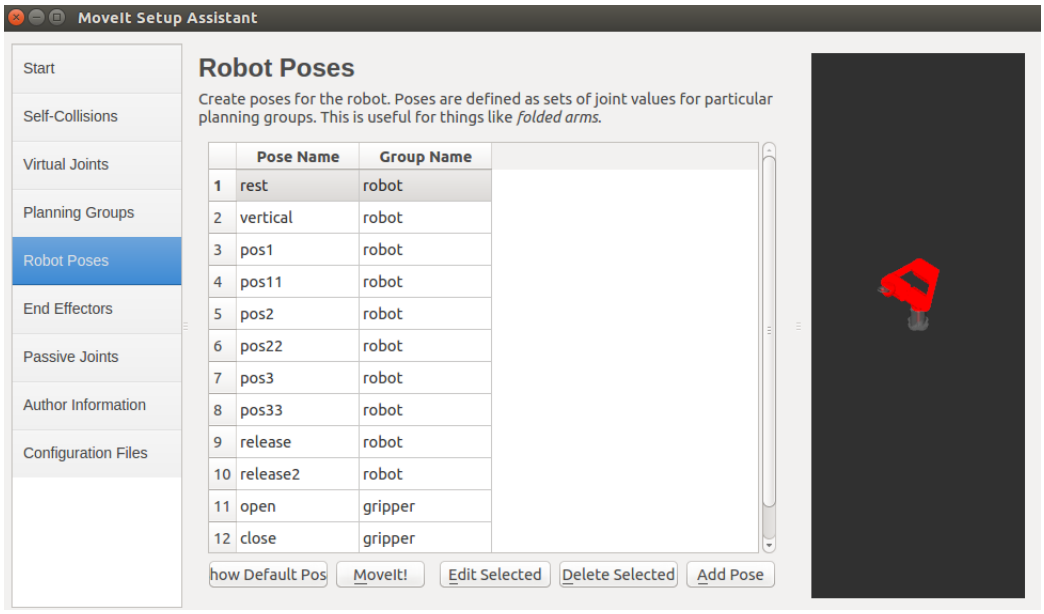


8.3 Irudia: *Ardatz-taldeen definizioa*

Ardatz-taldeak definitu ostean, robotaren poseak (robotaren ardatz bakoitzarentzako posizio espezifikoak) definitu daitezke. Pose hauek, ardatz-talde bakoitzeko ardatzen posizioak gordetzen dituzte, softwareak exekuta ditzan. Pose hauek kodean bertan definitu ditugun arren, laguntzailearen bitartez ere defini daitezke. 8.4 irudian kodean diseinatutako pose guztiak azaltzen dira, eskuinaldean posean parte hartzen duten ardatzak gorriz azaltzen direlarik.

Hurrengo pausua bukaerako eragingailuen definizioan datza. Pausu hau laguntzailearekin egin daitekeen arren, proiektu honetan eskuz kodetu da zati hau (geroago ikusiko dugun bezala). Bukaerako eragingailu bat definituz, gure beso robotikoan manipulatzeko balio duten zatiek (pintzek) estatus berezi bat hartzen dute, aparteko lotura bezala definitzen dira, eta beraiei eragiteko metodo bereziak egongo dira.

Hurrengo bi pausuak ere (*Passive Joints*, *Author Information*) alde batetara utziko ditugu, beraien konfigurazioa oso sinplea baita: lehenengoan inongo eragingailurik ez duten



8.4 Irudia: *Robotaren poseen konfigurazioa*

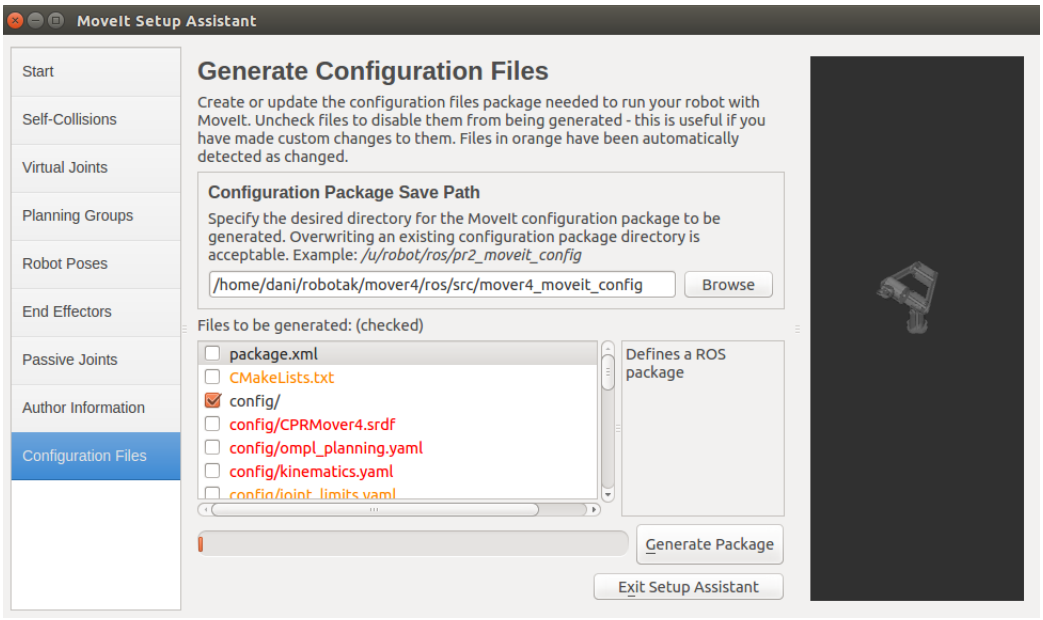
ardatzak definitzeko balio dute (gure kasuan, pintzaren oinarria) eta bigarrengoa egileari buruzko informazioa azaltzeko balio du.

Azkeneko pausua automatikoki sortzen diren konfigurazio fitxategi eta *.launch* fitxategi guztien sorrerarena da. Behin guztia ondo konfiguratuta dagoenean, sortuko den paketearen izena eta *Generate Package* aukerari ematea besterik ez dugu egin behar (ikus: 8.5 irudia).

8.3 SRDF fitxategia

8.3.1 Sarrera

5. kapituluan esan bezala, SRDF (Semantic Robot Description Format) fitxategiak URDF fitxategien luzapena balira bezala har daitezke. Fitxategi hauek URDF fitxategien bitartez adierazi ahal ezin diren elementu batzuk erazagutzea ahalbidetzen digute. Gure kasuan SRDF fitxategiaren eskeletoa Moveit!-eko laguntzailean egin arren, hainbat aldaketa egin behar izan genituen.Hurrengo ataletan SRDF fitxategi honen zatirik garrantzitsuenak azalduko dira.



8.5 Irudia: *Fitxategien sorrera Moveit!-eko laguntzailean*

8.3.2 Ardatz-taldeak

Aurreko kapituluetan esan bezala, ardatz-talde bat planifikatuko den mugimenduan parte hartzen duten robotaren atalak izango dira. Gure kasu partikularrean, mugimenduak soilik besoaren pintzak ez diren beste ardatz guztiek deskribatzea nahi dugu. Pintzak aparteko talde batean sartuko ditugu, hauen helburu bakarra pieza hartu eta uztearena baita.

Bi talde horiek erazagutzeko <*group*> etiketa erabiliko dugu. Lehenik eta behin beso robotikoaren artikulazioak (pintzenak ezik) *robot* izeneko taldean sartuko ditugu. Lana errazteko, SRDF fitxategien <*chain*> etiketa erabiliko dugu. Etiketa honen bitartez, artikulazio batetik hasita beste bateraino (biak barne) tarteko artikulazio guztiak talde berean sartuko ditu. Gure kasuan, *base_link* eta *gripperBody* roboteko atalen arteko artikulazio guztiak sartuko ditugu *robot* ardatz-taldean (ikus: 8.1 kodea).

8.1 Adibidea: "robot"ardatz-taldearen definizioa

```
1 <group name="robot">
2       <chain base_link="base_link" tip_link="gripperBody" />
3 </group>
```

Beste aldetik, pintza osatzen duten artikulazio guztiak *gripper* ardatz-taldean sartu ditugu. Kasu honetan *group* etiketa erabiltzeaz gain *joint* etiketa ere erabili dugu. Etiketa honen

bitartez ardatz-talde baten barneko artikulazio guztiak deini daitezke. Kasu honetan, pintza osatzen duten bi behatzak (*Gripper1* eta *Gripper2*) sartu ditugu *gripper* taldean (ikus: 8.2 kodea).

8.2 Adibidea: "gripper"ardatz-taldearen definizioa

```
<group name="gripper">
    <joint name="Gripper1" />
    <joint name="Gripper2" />
 </group>
```

8.3.3 Bukaerako eragingailuak

Bukaerako eragingailuak objektuak manipulatzeko edota beraiei eragiteko egindako roboten zatiak dira. Elementu berezi hauek ere defini daitezke SRDF fitxategietan. Elementu hauek definitzeko <*end_effector* etiketa erabili behar da. Gure kasuan Mover4 beso robotikoaren pintza izango litzateke bukaerako eragingailua. Berau definitzeko hainbat modu egon arren, guk erabilitakoa 8.3 kode-zatian ikus dezakeguna da.

8.3 Adibidea: Bukaerako eragingailuaren definizioa

```
<end_effector name="gripper" parent_link="gripperBody" group="robot"/>
```

8.3 kodean ikus dezakegun bezala, bukaerako eragingailu bat definitzeko, bere izena jartzeaz gain, eragingailua menpe dagoen lotura (*parent_link*) ere adierazi behar zaio. Beso robotikoaren pintzak *gripperBody* izeneko loturaren gainean eraikita daudenez, honakoa izango da *parent_link* eremuan jarri beharko duguna. Honez gain, *parent_link*-aren ardatz-taldearen izena ere jarri behar dugu, kasu honetan *robot* ardatz-taldea.

8.3.4 Robotaren poseak

8.2 atalean azaldu dugun Moveit!-eko laguntzailearen bitartez poseak defini daitezkela esan arren, gure proiektu honetan pose hauek kode bitartez erazagutu direla esan genuen. Robotarentzat aurredefinitutako poseak ere SRDF fitxategien bitartez erazagutu daitezke. Gure kasuan 10 pose desberdin erazagutu ditugu:

- 2 pose pieza lehenengo posizioan hartzeko (pintza bi posizio desberdinetan).

- 2 pose pieza bigarrengo posizioan hartzeko (pintza bi posizio desberdinetan).
- 2 pose pieza hirugarren posizioan hartzeko (pintza bi posizio desberdinetan).
- 2 pose, piezak uzteko posizioan (pintza bi posizio desberdinetan).
- Robota modu bertikalean jartzeko posea.
- Robota atseden posizioan jartzeko posea.

Pose hauek guztiak modu berean erazagutzen direnez, robota modu bertikalean jartzeko posea definitzen dugu 8.4 kodean.

8.4 Adibidea: Pose baten definizioa

```
<group_state name="vertical" group="robot">
       <joint name="Joint0" value="0" />
       <joint name="Joint1" value="0" />
       <joint name="Joint2" value="0" />
       <joint name="Joint3" value="0" />
</group_state>
```

Adibidean ikusten dugun bezala, pose bat definitzeko <*group_state*> etiketa erabiliko dugu. Pose bakoitza zein ardatz-taldeetan aplikatzekoa den ere adierazi beharko dugu. Gure kasuan pose guztiak *robot* ardatz-taldearentzat definitu ditugu. Behin hori guztia adieraztita, ardatz-taldeko ardatz bakoitzaren bukaerako posizioa adierazi beharko dugu (radianetan).

8.3.5 Kolisio matrizea

Moveit!-eko laguntzaileak kolisio-matrizeko hainbat zati automatikoki sortu arren, derrigorrezkoa bilakatzen da erabiltzaileak matrize honi lerro gehiago gehitzea. 5.2 atalean SRDF fitxategiei buruz aritu ginenean kolisio-matrizea definitzerakoan 3 talka-mota erazagutu daitezkeela azaldu genuen: *User*, *Adjacent* eta *Never*. Gure beso robotikoaren kolisio matrizea sortzerakoan honakoak definitu genituen:

- *User* motakoak:
  Moveit!-eko laguntzaileak robotaren URDF fitxategia aztertzean detektatutako talka posibleak dira hauek. Moveit!-eko laguntzaileak 9 talka posible detektatu zituen 8.5 adibideko kodean ikus dezakegun bezala.

8.5 Adibidea: "User"motako talkak kolisio matrizean

```
<disable_collisions link1="base_link" link2="geoJoint2" reason="User" />
<disable_collisions link1="base_link" link2="geoJoint3" reason="User" />
<disable_collisions link1="geoJoint1" link2="geoJoint3" reason="User" />
<disable_collisions link1="geoJoint2" link2="geoJoint4" reason="User" />
<disable_collisions link1="geoJoint2" link2="gripperFinger1" reason="User" />
<disable_collisions link1="geoJoint2" link2="gripperFinger2" reason="User" />
<disable_collisions link1="geoJoint3" link2="gripperFinger2" reason="User" />
<disable_collisions link1="geoJoint4" link2="gripperFinger1" reason="User" />
<disable_collisions link1="geoJoint4" link2="gripperFinger2" reason="User" />
```

- *Adjacent* motakoak:
  Erabiltzaileak definitzen dituen talka posibleak dira. Talka hauen arrazoia bi ardatzak bata bestearen ondoan daudela izango da. Mota honetako 7 talka posible azaldu ditugu 8.6 adibideko kodean azaltzen den bezala.

8.6 Adibidea: "Adjacent"motako talkak kolisio matrizean

```
<disable_collisions link1="base_link" link2="geoJoint1" reason="Adjacent" />
<disable_collisions link1="geoJoint1" link2="geoJoint2" reason="Adjacent" />
<disable_collisions link1="geoJoint2" link2="geoJoint3" reason="Adjacent" />
<disable_collisions link1="geoJoint3" link2="geoJoint4" reason="Adjacent" />
<disable_collisions link1="geoJoint4" link2="gripperBody" reason="Adjacent" />
<disable_collisions link1="gripperBody" link2="gripperFinger1" reason="Adjacent" />
<disable_collisions link1="gripperBody" link2="gripperFinger2" reason="Adjacent" />
```

- *Never* motakoak:
  Mota honetako talkek ezinezkoak direnak adierazteko balio dute. Gure kasuan mota honetako 3 definitu ditugu (ikus: 8.7 adibideko kodea).

8.7 Adibidea: "Never"motako talkak kolisio matrizean

```
<disable_collisions link1="geoJoint2" link2="gripperBody" reason="Never" />
<disable_collisions link1="geoJoint3" link2="gripperBody" reason="Never" />
<disable_collisions link1="geoJoint3" link2="gripperFinger1" reason="Never" />
```

8.4 Konfigurazio fitxategiak

8.4.1 Sarrera

SRDF fitxategiaz gain beste hainbat konfigurazio fitxategi behar ditu Moveit! softwareak funtzionatzeko. Fitxategi hauek Moveit!-eko laguntzailearekin eraikitako paketearen barruan daude, *config* direktorioaren barruan. Hurrengo ataletan oinarrizkoak zein proiektu honetan erabiltzeko beharrezkoak izan diren konfigurazio fitxategiak azalduko dira.

8.4.2 *controlers.yaml* fitxategia

Fitxategi honetan beso robotikoa Moveit!-en bitartez kontrolatzeko *FollowJointTrajectory* zerbitzaria zein izango den eta kontrolatu beharreko artikulazioen izenak adieraziko dira. 8.8 adibideko kodean Mover4 beso robotikoa kontrolatzeko *controlers.yaml* fitxategia ikus dezakegu.

8.8 Adibidea: Mover4 besoaren "controlers.yaml"fitxategia

```
controller_list:
 - name: cpr_mover
   action_ns: follow_joint_trajectory
   type: FollowJointTrajectory
   default: true
   joints:
     - Joint0
     - Joint1
     - Joint2
     - Joint3
     - GripperFixed
     - Gripper1
     - Gripper2
```

8.4.3 *fake_controllers.yaml* fitxategia

Fitxategi honen bidez robot erreala kontrolatu behar ez dugunean, hau da, bistaratzaile baten bidez planifikatutako mugimenduak ikusi nahi ditugunean Moveit!-ek mugimendu horiek simulatzeko erabiliko dituen kontroladoreak adieraziko dira. Bertan, kontroladore faltsu (*fake*) hauek mugituko dituzten artikulazioak azalduko dira (ikus: 8.9 adibideko kodea).

8.9 Adibidea: Mover4 besoaren "fake_controllers.yaml"fitxategia

```
controller_list:
  - name: fake_robot_controller
    joints:
      - Joint0
      - Joint1
      - Joint2
      - Joint3
```

Ikus dezakegunez, pintza ez dugu kontrolatuko kontroladore faltsuen bidez, aurretik esan bezala, pintza beste metodo batzuen bidez kontrolatuko dugulako.

8.4.4 *joint_limits.yaml* fitxategia

joint_limits.yaml fitxategiaren bidez robotaren URDF fitxategian adierazitako mugak (posizio, abiadura, esfortzu maximoak...) kontuan hartzeko adieraziko diogu Moveit! softwareari. 8.10 adibideko kodean Mover4 beso robotikoaren URDF fitxategian mugak dituzten artikulazioak kontuan hartzeko adierazten dugu.

8.10 Adibidea: Mover4 besoaren "joint_limits.yaml"fitxategia

```
joint_limits:
  Joint0:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 30
    has_acceleration_limits: false
    max_acceleration: 0
  Joint1:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 100
    has_acceleration_limits: false
    max_acceleration: 0
  Joint2:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 100
    has_acceleration_limits: false
    max_acceleration: 0
  Joint3:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 100
    has_acceleration_limits: false
    max_acceleration: 0
```

```
Gripper1:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 100
    has_acceleration_limits: false
    max_acceleration: 0
  Gripper2:
    has_position_limits: true
    has_velocity_limits: true
    max_velocity: 100
    has_acceleration_limits: false
    max_acceleration: 0
```

8.4.5 *kinect.yaml* fitxategia

Fitxategi honetan Kinect sentsorea Moveit!-ekin batera erabiltzeko konfigurazioak azaltzen dira. Fitxategi honi buruz 8.6 atalean arituko gara.

8.4.6 *kinematics.yaml* fitxategia

8.11 adibideko kodean Mover4 besoaren Moveit!-eko konfigurazio fitxategien artean aurkitzen den *kinematics.yaml* fitxategia dugu. Fitxategi honetan alderantzizko zinematika aplikatzerakoan erabiliko den ebazlea eta honen parametroak agertzen dira. Gure kasuan *kdl* ebazlea aukeratu da, denetatik eraginkorrena eta azkarrena baita. Mugimenduak kalkulatzerakoan, 0.005-eko bereizmen eta timeout-ak ezarri dizkiogu (denbora horren barruan inongo erantzunik ez badu ematen mugimenduaren planifikazioa bertan behera utziko dugu). Gainera, mugimendu bat planifikatzerakoan hiru saiakera emango dizkiogu ebazleari. Hiru saiakera ostean ez badu soluziorik topatzen, mugimendua modu seguruan exekutatzea ezinezkoa dela esango dugu.

8.11 Adibidea: Mover4 besoaren "kinematics.yaml"fitxategia

```
robot:
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.005
  kinematics_solver_attempts: 3
```

8.4.7 *ompl_planning.yaml* fitxategia

Konfigurazio fitxategi hau Moveit!-eko laguntzaileak sortzen du modu automatikoan. Fitxategi honetan mugimenduak planifikatzerako orduan softwareak erabiliko dituen parametroak azaltzen dira.

8.5 *.launch* fitxategiak

8.5.1 Sarrera

Moveit! softwarea hainbat eta hainbat exekutagarri (*.launch* fitxategi) zein nodoren menpe dago. Hori dela eta laguntzailea exekutatu ondoren *.launch* fitxategi asko sortzen ditu. Fitxategi hauen azalpena konplikatu ez dadin, softwareak behar dituen fitxategi guztiak exekuzioan jartzen dituen *demo.launch* fitxategia azalduko dugu.

8.5.2 *demo.launch* fitxategia

Aurretik esan bezala, fitxategi honek (ikus: 8.12 adibideko kodea) Moveit!-ek funtzionatzeko beharrezko fitxategi guztiak jarriko ditu martxan. Fitxategi hau automatikoki sortzen den arren, hainbat aldaketa egin behar zaizkio erabiltzen den gainerako softwarearekin lan egin dezan. Ikus dezakegun bezala, hasteko, Mover4 beso robotikoaren kontroladorea jarriko du martxan. Hau beharrezkoa izango da benetako robota mugitu nahi dugunean.

Honez gain, fitxategi honek *planning_context.launch* fitxategia martxan jarriko du. Fitxategi honek mugimenduak planifikatzeko beharrezko fitxategi guztiak jartzen ditu martxan.

Honen ondoren, Moveit! softwarean garrantzitsuenetakoa den interfazea exekutatzen jarriko du. Interfaze hau *move_group* interfazea da. Honi esker konfigurazio fitxategietan erazagututako artikulazioak kontroladoreen bitartez (faltsuak zein errealak) eragin ahal izango ditu softwareak. Honez gain, robota funtzio berezi batzuen bitartez kontrolatzea ahalbidetuko digu (honi buruz 8.8 atalean jardungo dugu).

Azkenik. rViz bistaratzailea abiatuko du fitxategi honek. Modu honetan planifikatzen ditugun mugimenduak ikusi ahal izango ditugu.

.launch fitxategi honen bitartez martxan jartzen ditugun fitxategiek menpeko dituzten fitxategiak jartzen dituzte martxan, hau da, egindako konfigurazio guztiak *demo.launch* fitxategi honen bitartez aplikatzen dira.

8.12 Adibidea: demo.launch fitxategia

```
<launch>

 <param name="robot_type" value="mover4"/>

 <node name="cpr_mover" pkg="$(find cpr_mover)" type="cpr_mover"/>

 <include file="$(find mover4_moveit_config)/launch/planning_context.launch">
    <arg name="load_robot_description" value="true"/>
 </include>


 <include file="$(find mover4_moveit_config)/launch/move_group.launch">
    <arg name="allow_trajectory_execution" value="true"/>
    <arg name="fake_execution" value="false"/>
    <arg name="info" value="true"/>
    <arg name="debug" value="$(arg debug)"/>
 </include>

 <include file="$(find cpr_rviz_plugin)/cpr_mover4.launch"> </include>

</launch>
```

8.6 Moveit! Kinect sentsorearekin

8.6.1 Sarrera

Aurreko hainbat kapitulutan esan bezala, gure proiektuaren helburuak lortzeko ezinbesteko erraminta dugu Kinect sentsorea. Kapitulu honetan aztergai dugun Moveit! softwareak hainbat sentsore moten barneraketa ahalbidetzen du, horretarako interfaze bat eskeiniz. Atal honetan Kinect sentsorearen barneraketa egiteko egin beharreko konfigurazio guztiak azalduko dira.

8.6.2 *Pointcloud-aren filtroa*

Proiektuaren helburua soilik oztopoak identifikatzea ez denez, baizik eta manipulatu beharreko piezaren posizioa zehaztu behar dugun, Kinect-aren *pointcloud*-a filtratzea beharrezkoa izango da.

Filtro honen helburua bikoitza izango da: hasteko, manipulatu beharreko piezaren posizioa jakingo dugu, eta, bestalde, Moveit!-ek pieza hori ez du oztopotzat kontuan hartuko (oztopotzat identifikatuko balu ez zukeen hartuko).

Hori lortzeko, ROSeko beste pakete berri bat sortu zen: *pointcloud2_filter*. Bere baitan aurkitu dezakegun *pointcloud2_filter.cpp* kodean Kinect sentsoreak */camera/depth_registered/points* *topic*-ean publikatzen duen *pointcloud*-a hartu eta piezaren koloreko puntuak bertatik kenduko ditu. Modu honetan dagoeneko Moveit! sentsoreak ez ditu piezak oztopotzat hartuko. Filtratutako *pointcloud*-a *output topic*-ean publikatuko da hurrengo atalean azalduko den konfigurazio fitxategiaren bitartez Moveit!-ek barnera dezan.

8.6.3 Konfigurazio fitxategia: *kinect.yaml*

Sentsore desberdinentzako konfigurazio desberdina egin behar izango litzatekeen arren, honakoan Microsoft-eko Kinect sentsorea konfiguratzeko egin beharrekoa azalduko dugu.

3D kamera honen kasuan, berauk informazioa publikatzen duen *topic*-ak azaldu beharko dizkiogu, berau kontrolatzeko pluginaz gain. Gure kasuan softwarean barneratu nahi dugun informazio bakarra filtratutako puntu-lainoa (*pointcloud*-a) izango da (*output topic*-ean).

Honez gain, 3D kameren kasuan, Moveit! softwareak *self-filtering* deritzon filtroa aplikatzen dio ematen diogun *pointcloud*-ari. Filtro honen bitartez ingurunean dagoen robota ere ez du hartuko oztopotzat bere ikus-eremuan sartzen bada. Hori lortzeko, Moveit! softwareari emandako URDF fitxategia kontuan hartzen du eta unean ikusten duen irudiarekin konparatzen du. URDF-arekin bat etortzen den guztia ez du oztopotzat hartuko. Filtro hau konfiguratzeko *kinect.yaml* fitxategian azaltzen zaizkigun *padding_offset* eta *padding_scale* parametroak ditugu. Bi parametro hauen arteko oreka topatuz robota ez da oztopoa balitz bezala tratatuko, piezarekin egin dugun modu berean.

Oztopoen detekzioa zein distantziaraino egin nahi den adierazi beharko dugu *max_range* parametroaren bitartez. Oztopoen detekzioak baliabide asko eskatzen dituenez eta piezak

sentsoretik 1.2 metrotara baino hurbilago egongo direnez, distantzia hortatik urrunago dauden oztopoak ez ditu detektatuko.

Azkenik, *self-filtering* teknika aplikatu ostean lortutako *pointcloud*-a zein *topic*-etan argitaratuko den adierazi beharko dugu. Gure kasuan topic honi *output_cloud* deitu diogu.

8.13 Adibidea: kinect.yaml konfigurazio fitxategia

```
sensors:
- sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
  point_cloud_topic: /output
  max_range: 1.2
  point_subsample: 1
  padding_offset: 0.05
  padding_scale: 1.0
  point_subsample: 1
  filtered_cloud_topic: output_cloud
```

8.6.4 Integrazioa Moveit!-ekin: *CPRMover4_moveit_sensor_manager.launch*

Kinect sentsorearen oztopoen detekzioa konfiguratuta, berau Moveit! softwarean integratu beharko dugu. Horretarako, aurretik aipatutako *demo.launch* fitxategiak kargatzen duen *CPRMover4_moveit_sensor_manager.launch* fitxategia sortu beharko dugu.

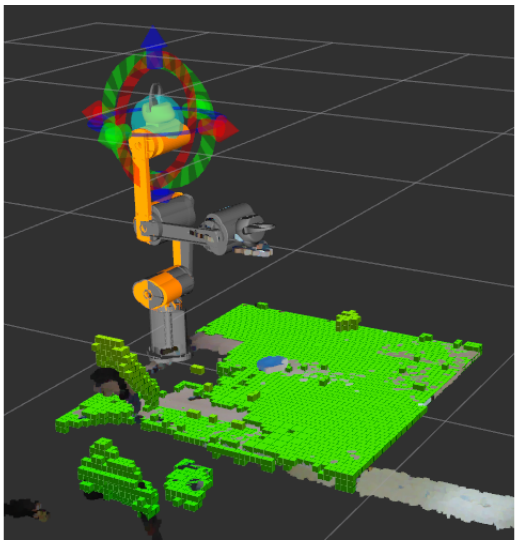
Fitxategi honetan, aurreko atalean garatutako *kinect.yaml* fitxategia kargatzeko adieraziko diogu softwareari (ikus: 8.14 adibideko kodea).

8.14 Adibidea: CPRMover4_moveit_sensor_manager.launch konfigurazio fitxategia

```
<launch>
    <rosparam command="load" file="$(find mover4_moveit_config)/config/kinect.yaml" />
</launch>
```

8.7 Moveit! rViz bistaratzailean

Moveit!-eko konfigurazio hauek guztiak egin ondoren, aurreko kapituluko 7.5 irudian agertzen zenaz gain Moveit!-eko kontrol panela azaltzeko *MotionPlanning* elementua gehitu beharko diogu rViz bistaratzaileari. Honela, gure beso robotikoarentzat mugimenduak planifikatzen has gaitezke, eta oztopoen detekziorako sistema martxan jarriko da 8.6 irudian ikus dezakegun bezala.

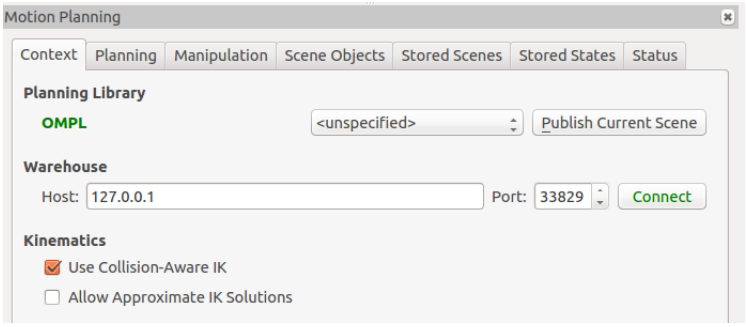


8.6 Irudia: *Moveit! softwarea rViz bistaratzailean*

rViz bistaratzailean detektatutako oztopoak "kubotxo"batzuen bidez adierazten direla ikusten da. Gainera, manipulatu beharreko pieza (urdinez) eta robota oztopotzat hartzen ez direla ikus dezakegu.

8.7.1 Mugimenduen planifikazioa rViz-en

Mugimenduen planifikazio guztia rViz-en gehitu dugun Moveit!-eko kontrol panelaren bitartez gauzatzen da (ikus: 8.7 irudia). Panel honen bitartez hainbat gauza desberdin egin daitezke:

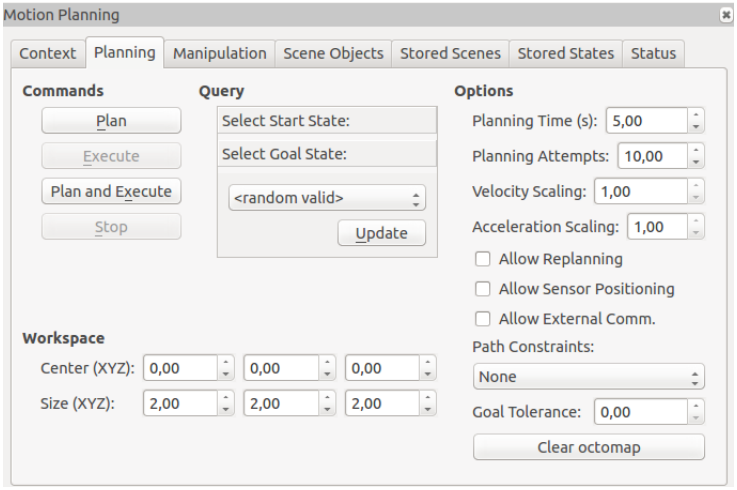


8.7 Irudia: *Moveit!-en kontrol panela*

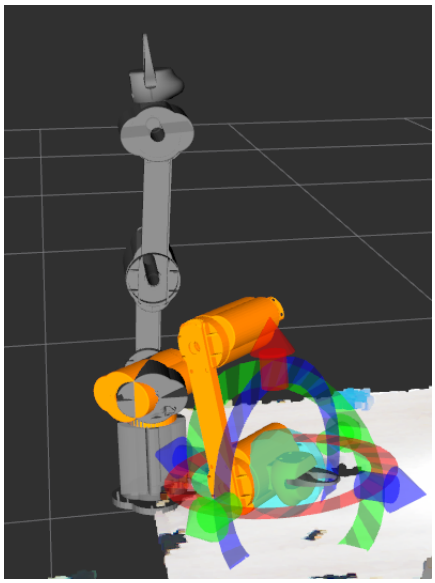
- Mugimendu planifikazioaren parametroak editatu:
  Moveit!-eko kontrol panelaren *Context* atalean mugimenduak planifikatzeko erabi-

liko den liburutegia espezifikatu daiteke. Horren arabera kalkulatutako mugimenduak desberdimak izango dira: azkarragoak, optimoagoak, zehatzagoak...

- Robotaren poseak aldatu:
  Planning atalean (ikus: 8.8 eta 8.9 irudiak) aurretik definitutako poseak exekutatu daitezke. Poseen arteko mugimendua ere planifikatuko du eta rViz bistaratzailean mugituko da robota. Honez gain, beste hainbat parametro alda daitezke, hala nola, planifikaziorako denbora edota saiakera maximoak.



8.8 Irudia: *Moveit!-en kontrol panelaren Planning atala*



8.9 Irudia: *Poseen exekuzioa. Irudian, laranja kolorez, bukaerako posea*

- Objektuak identifikatu eta manipulatu:
  Proiektu honetan Moveit!-eko gaitasun hauek erabili ez ditugun arren, softwarea piezak identifikatzeko eta manipulatzeko gai da. *Scene Objects* atalaren bitartez objektuak identifikatu eta *Manipulation* atalarekin pieza horiek manipulatu ahal izango genituzke.

- Inguruneak kargatu/gorde:
  rViz-en bistaratzeko ingurune desberdinetan Moveit! erabiltzea gaitasun interesgarria izan daiteke. Kontrol paneleko *Stored Scenes* atalarekin hainbat ingurune desberdin kargatu eta gorde ditzakegu probak egiteko.

- Robotaren egoerak kargatu/gorde:
  Ingurunetaz gain, Moveit!-ek robotaren egoera desberdinak kargatzen eta gordetzen uzten ditu, probak egiteko. Hori egiteko kontrol paneleko *Stored States* atalera jo beharko dugu

Proiektu honetan soilik mugimenduaren planifikaziorako *Planning* atala erabiliko dugu probak egiterakoan. Hurrengo atalean piezen manipulazioari buruz arituko gara.

9. KAPITULUA

Egiaztapena: piezen manipulazioa

9.1 Sarrera

Proiektuaren helburu nagusia, piezen manipulazioa, honako kapituluan azalduko diren konfigurazioekin eta garatutako kodearekin lortutakoa da. Behin hori lortuta, manipulazioa oztopoak ekidinez lortzeko Moveit! softwarearekin batera erabiliko da.

Hurrengo ataletan piezen manipulaziorako garatu beharreko piezen detekzioa eta robota mugitzeko inplementatutakoa azalduko dira.

Honez gain, proiektuaren garapenean zehar egindako probak azalduko dira (simulazioen bitartez zein errealitatean egindakoak).

9.2 Piezen posizioaren identifikazioa

9.2.1 Sarrera

Piezen manipulazioa egiteko lehenengo pausua pieza non dagoen jakitea da. Honako helburua lortzeko ere Kinect sentsorea erabiliko dugu, kasu honetan, sentsorearen koloretako kameraren 2D irudia. Irudi hori segmentatuz, piezaren kolorea identifikatuz, honen posizioa lortuko dugu. Behin hori jakinda, robota mugitzen has gaiteke.

9.2.2 Irudien segmentazioa: *segment_blob* paketea

Irudiaren segmentazioa egiteko, RSAIT taldeak eskainitako *segment_blob* paketea erabili da. Honako paketearen bitartez, identifikatu beharreko piezaren kolorea adieraziz, pieza horren zentroidea kalkulatu eta bere posizioa adieraziko digu.

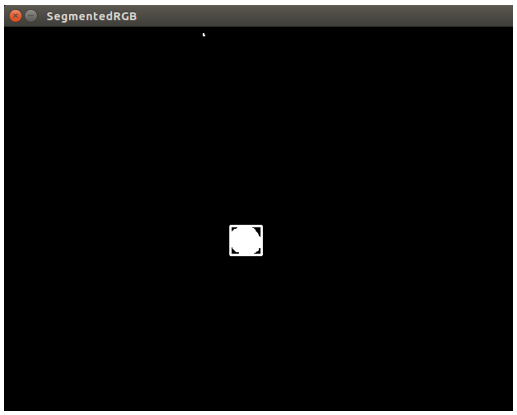
Lehenengo pausua emandako paketea proiektuari egokitzea izan zen. Hasteko, identifikatu beharreko kolorea aldatu zen, gure piezaren kolore urdinarekin bat etortzeko. Horretarako, paketeko *segment_blob.cpp* kodean identifikatu beharreko kolorea adierazi zen, 9.1 adibidean agertzen den moduan.

9.1 Adibidea: Piezaren kolorearen erazagutzea segment_blob.cpp programan

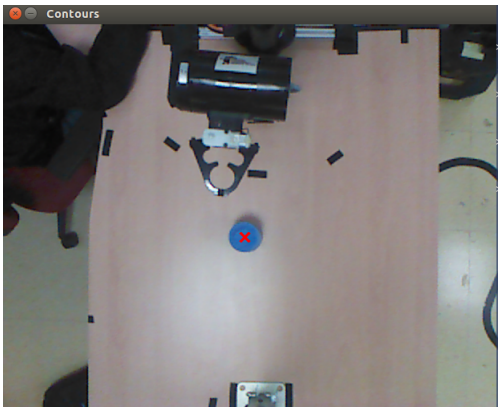
```
lowR = 209;
highR = 255;
lowG = 102;
highG = 255;
lowB = 0;
highB = 105;
```

Programa honi kolore zehatza adierazi ezean, RGB kolore sistemako kolore bakoitzerako mugak ezarri ziren. 9.1 adibideko kodean azaltzen den moduan, adibidez, irudiko puntu baten kolore gorria (R) 209 eta 255 balioen artean badago, berdea (G) 102 eta 255 artean, eta urdina (B) 0 eta 105 artean, gure piezaren kolorea dela identifikatuko dugu.

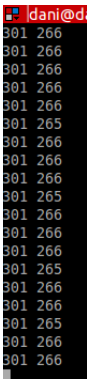
Irudian irizpide horiek biltzen dituzten puntuak hartuta (9.1 irudian), hauen zentroidea aurkituko da eta honen x eta y koordenatuak emango ditu (ikus: 9.2 eta 9.3 irudiak).



9.1 Irudia: *Irizpideak jarraitzen dituzten puntuak*



9.2 Irudia: *Piezaren zentroidea, irudian, gurutze gorri batekin*



9.3 Irudia: *Piezaren x eta y koordenatuak*

Zentroidea kalkulatu ostean dagoeneko jakingo dugu pieza inguruneko hiru posiziotik zeinetan dagoen. Kalkulatutako zentroidearen x posizioaren arabera jakingo dugu posizioa. Horretarako x balioen tarte batzuk zehaztu ziren (ikus: 9.2 adibideko kodea).

9.2 Adibidea: Piezaren posizioaren estimazioa segment_blob.cpp programan

```
1  if(cres.x > 400){
2     position->data = "three";
3     posPub.publish(position);
4   }else if(cres.x < 400 && cres.x > 255){
5         position->data = "two";
6     posPub.publish(position);
7
8   }else{
9         position->data = "one";
10    posPub.publish(position);
11  }
```

Adibidean azaltzen diren *one*, *two* eta *three String*-ak *position topic*-ean argitaratuko dira, hurrengo atalean azalduko dugun programan erabiliak izateko.

9.3 *move_group* interfazea

9.3.1 Sarrera

Move_group Moveit! softwareak robotak programa bidez kontrolatzeko duen interfazea da. Interfaze honekin hainbat programazio lengoientzat eginiko funtzioak inplementatzen dira mugimenduak planifikatzeko eta exekutatzeko. Gure kasuan piezak manipulatzerako orduan pose desberdinen artean mugitzeko eta pintzei eragiteko erabiliko dugu.

9.3.2 Mover4 beso robotikoa mugitzen *move_group* interfazearen bidez

Atal honetan proiektuan egindako *move_group* interfazeko inplementazioaren zatirik garrantzitsuenak azalduko ditugu. Bertan, piezaren posizioa eskuratzen denetik berau piezak uzteko posiziora eraman arteko pausu guztiak adieraziko dira.

move_group interfazearen inplementazio hau *mover4_moveit_config* paketeko *proba.cpp* programan arkituko dugu.

Posizioaren identifikazioa

9.2.2 atalean azaldutako *segment_blob.cpp* programan piezaren posizioa *position topic*-ean argitaratzen zela azaldu genuen. Garatutako *proba.cpp* programan *topic* horretara harpidetuta dagoen *subscriber* bat sortu da (ikus: 9.3 adibideko kodea).

9.3 Adibidea: Piezaren posizioaren estimazioaren jasotzea proba.cpp programan

```
1  objPos = nh_.subscribe<std_msgs::String>("/segment_camera_image/position", 1,objPosCallback );
```

Subscriber honek *position topic*-eko *String*-a hartu eta honen araberako exekuzioa jarriko du martxan, hurrengo ataletan ikusiko dugun bezala.

Besoaren mugimenduaren kontrola

Piezaren posizioaren arabera, besoaren loturek posizio desberdin bat hartu beharko dute, leku desberdin batera mugitu beharko da besoa. Hori dela eta, posizioaren mezu bat ailegatzen den bakoitzean loturak hartu beharko dituzten posizioak esleitzen dira 9.4 adibideko kodean agertzen den moduan.

9.4 Adibidea: Robotaren loturen posizioak piezaren posizioaren arabera

```
if(objectPosition == "one"){
        group1_variable_values[0] = -0.7416;
        group1_variable_values[1] = 1.0365;
        group1_variable_values[2] = 1.1429;
        group1_variable_values[3] = 0.7846;
    }else if(objectPosition == "two"){
        group1_variable_values[0] = -0.00879;
        group1_variable_values[1] = 1.0365;
        group1_variable_values[2] = 1.1429;
        group1_variable_values[3] = 0.7846;
    }else{
        group1_variable_values[0] = 0.5409;
        group1_variable_values[1] = 1.0365;
        group1_variable_values[2] = 1.1429;
        group1_variable_values[3] = 0.7846;
    }

    group1.setJointValueTarget(group1_variable_values);
```

Loturen posizio horiek esleitu ostean, uneko posiziotik bukaerako poserainoko mugimendua planifikatuko dugu. Horretarako *move_group* interfazeak *plan* funtzioa du (ikus: 9.5 kodea).

9.5 Adibidea: Mugimenduen planifikazioa plan funtzioarekin

```
    moveit::planning_interface::MoveGroup::Plan my_plan;
    success = group1.plan(my_plan);
```

Sofwareak 3 eta 5 segundu artean beharko ditu mugimendua planifikatzeko. Posea aldatzeko tarteko mugimenduak kalkulatutakoan, mugimenduak exekutatuko ditu 9.6 adibidean agertzen den funtzioa exekutatuz.

9.6 Adibidea: Mugimenduen exekuzioa lortzeko funtzioa

```
group1.move();
```

Hainbat mugimendu kalkulatu eta exekutatu behar ditugunez, garrantzitsua da mugimendua noiz burutu den jakitea. Hori jakin ezean posiblea izango litzateke mugimendu bat burutu aurretik hurrengoa planifikatzen hastea. Hori gertatuko balitz Moveit! softwarea gelditukо litzateke. Mugimendua burutu denentz jakiteko 9.7 adibideko kodean agertzen dena inplementatu da.

9.7 Adibidea: Mugimendua burutu denentz ziurtatzeko funtzioa

```
while(same == false){
        group1.getCurrentState()->copyJointGroupPositions(group1.getCurrentState()->getRobotModel
    ()->getJointModelGroup(group1.getName()), group1_actual_values);

        for(i=0;i<4;i++){
            if(group1_actual_values[i] < group1_variable_values[i]){
                same=false;
            }else{
                same=true;
            }
        }
    }
```

Adibidean ikus dezakegun bezala, hasteko, robotaren loturen uneko posizioa lortzen dugu bukaerako posizioarekin konparatzeko behin eta berriz. Posizioak beti berdin-berdinak ez direnez (10^{-3}-ko errorea dute), txikiagotasun erlazioa adierazi da beraien artean.

Behin mugimendua burututa dagoela ziurtatuta, hurrengo mugimenduak planifikatu eta exekutatu ahal izango dira.

Pintzaren mugimenduaren kontrola

Beso robotikoaren pintzari inongo posiziorik esleitu ezin zaizkionez (komando bidez funtzionatzen du soilik), komando horiek jasotzen dituen *topic*-era bidali beharko ditugu pintzarentzako mezuak.

Pintzaren kontrola edukitzeko *topic*-a *CPRMoverCommands* da. Bertan *GripperOpen* edo *GripperClose String*-ak bidaliz, pintza ireki edo itxiko dugu, hurrenez hurren.

Hori lortzeko, 9.8 eta 9.9 adibideko kodeetan agertzen dena inplementatu da.

9.8 Adibidea: Harpidetza CPRMoverCommands topic-era

```
gripperCommand =  nh_.advertise<std_msgs::String>("CPRMoverCommands", 1, true);
```

9.9 Adibidea: Pintza irekitzeko aginduaren bidalketa

```
command.data = "GripperOpen";
 gripperCommand.publish(command);
```

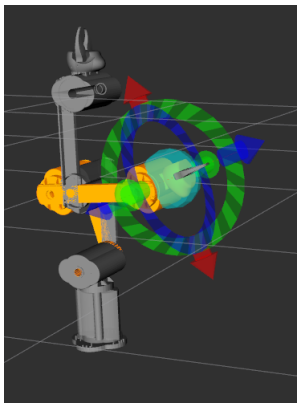
9.4 Piezen manipulazio-begizta

9.4.1 Sarrera

Atal honetan piezak manipulatzerako orduan aurreko atalean azaldutako *proba.cpp* programak darabilen begizta azalduko dugu (E3 eranskinean). Begizta nagusia infinitua da eta begizta bakoitzean lau mugimendu bereizten dira.

9.4.2 Pieza hartu aurretik

Piezaren posizioa lortu aurretik, robota atseden posean egongo da (ikus: 9.4 irudia). Posizioa jasotzen duenean, pintza irekiko du.

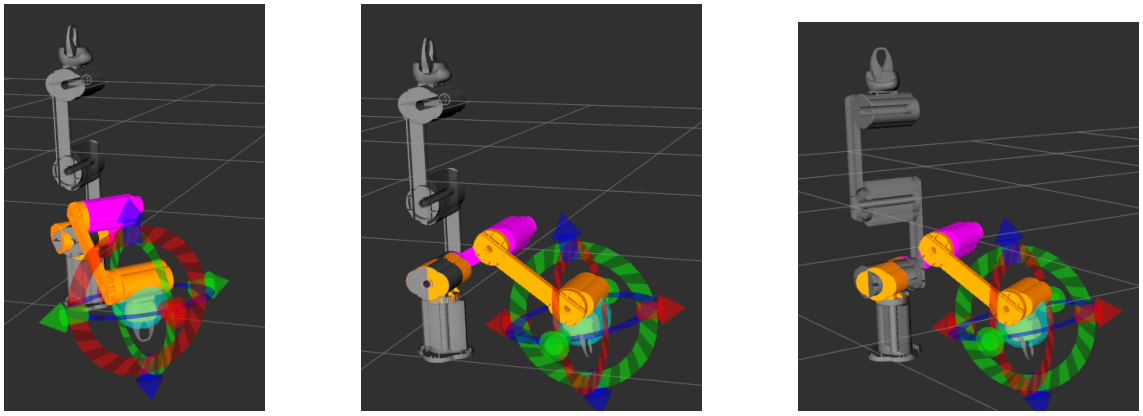


9.4 Irudia: *Beso robotikoa atseden posizioan*

9.4.3 Pieza hartzea

Behin piezaren posizioa jasota eta pintza irekita, piezaren posizioaren araberako posea lortzeko mugimendua planifikatuko du (ikus: 9.5 irudi-sorta). Mugimendua modu seguruan egitea posiblea dela badio (oztopoen kontra jo gabe eta bere loturen limiteak pasa

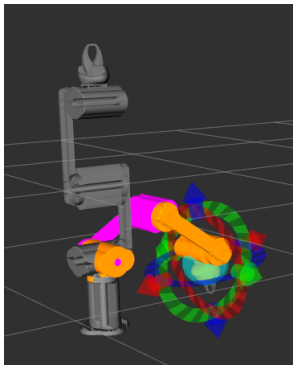
gabe), mugimendua burutuko du. Behin mugimendua burututa, pintza itxi eta pieza hartuko du.



9.5 Irudia: *Piezaren posizioaren araberako poseak*

9.4.4 Pieza uztea

Pieza hartu duenean, piezak uzteko posera joateko mugimenduak kalkulatuko ditu, 9.6 irudiak erakusten duenera. Mugimendua posible bada, mugitu egingo da. Mugimendua burutzen duenean, pintza berriro irekiko du.



9.6 Irudia: *Beso robotikoa piezak uzteko posizioan*

9.4.5 Atseden posiziora joaten

Pintza irekitzen duenean, hasieran zuen atseden posiziora (9.4 irudian agertzen dena) bueltatuko da eta pintza itxiko du. Hemendik aurrera beste posizio bat ailegatzen zaionean, aurretik azaldutako begizta guztia errepikatuko du.

9.4.6 Piezen manipulazioa: simulazioak eta errealitatea

Piezen manipulazioa gauzatzen duen begizta probatzerakoan, bi motatako probak egin dira: hasteko, moveit! softwareak inplementatzen duen simuladorea eta rViz bistaratzailearekin eta ondoren, benetako robotarekin.

Simuladorearen bidezko probak

Diseinatutako manipulazio-begizta errealitatean probatu aurretik moveit! softwareak inplementatzen duen simuladorea erabili da. Simuladore honekin batera, rViz bistaratzailea erabili da manipulazioa errealitatean nola garatuko litzatekeen ikusteko.

Proba hauetan, beso robotikoari pose zehatz batzuk exekutatzea agindu zitzaion, eta rViz bistaratzailearen bitartez pose hauen zehaztazuna aztertu zen.

Honez gain, aldez aurretik diseinatutako manipulazio-begizta behin eta berriz exekutatzea agindu zitzaion mover4 beso robotikoari, rViz-eko ingurunean pieza birtualak posizio desberdinetan jarriz, eta robotaren traiektorian oztopo birtualak erantsiz.

Simulazio hauen bitartez robotaren mugimenduen fintzea lortu zen. Mugimenduak leundu ziren eta traiektorien kalkulu-denbora eta hauen exekuzio-denbora murriztu zen moveit!-eko parametroak aldatuz: mugimenduen ebazlea, saiakera maximoen kopurua, mugimenduen zehaztasun maila...

Behin manipulazio-begiztak egin beharrekoa modu egokian egiten zuela aztertuta, eta traiektorien fintzea lortuta, errealitateko probak egitera pasa zen.

Probak errealitatean

Probak errealitatean egitera pasatzerakoan, moveit! softwareari dagoeneko robotaren kontroladore errealak (jada ez faltsuak edo *fake controllers* direlakoak) erabiltzeko agindu zitzaion. Honekin batera, softwarea Kinect sentsoreakin konektatu eta honek lortzen zituen irudiak segmentatzera bidali ziren, benetako piezaren posizioa hautemateko.

Errealitatean simulazioan egindako proba berak egin ziren. Hasteko, robotari aldez aurretik definitutako poseak exekutatzea agindu zitzaion. Behin pose horiek zehaztasunez exekutatzen zituela ikusita, manipulazio-begizta behin eta berriz exekutatzea agindu zitzaion.

Errealitateko probek simulazioaren antzerako bilakaera izan zuten, izan ere piezen manipulazio modu egokian garatzen zen, oztopoak tartean egon ala ez: piezen posizioa ondo hautematen zen, poseak modu zehatz eta egokian deskribatzen ziren eta begizta behin eta berriz exekutatu arren ere emaitza egokiak ematen zituen.

Hala ere, errealitatean, simulazioan kontuan hartzen ez ziren bi faktoreengatik batzuetan (hamar saiakeratik behin gutxi gorabehera) piezen manipulazioa ez zen modu egokian garatzen. Bi faktore horiek gelaren argitasuna eta pintzaren zehaztasuna ziren.

Gelaren argitasunak piezen posizioaren detekzioan eragiten zuen. Argi-maila desberdinetan, Kinect sentsorearen koloretako kamerak koloreak modu desberdinean hautematen zituen, eta, beraz, manipulatu beharreko piezaren kolorea ez zen modu egokian hautematen. Hori dela eta, piezaren posizioa ez zion ongi bidaltzen moveit! softwareari, eta piezaren manipulazioa ezinezko bihurtzen zen. Arazo honi aurre egiteko, instalazioaren gaineko argia uneoro piztuta uzten zen. Argi leun horrek koloreen pertzepzioan laguntzen zuen, argi-maila homogeneo bat mantenduz.

Pintzaren zehaztasuna, aldiz, konpontzeko ezinezkoa gertatu den arazo bat da. Moveit! softwareak beso robotikoari poseak zehatz bidali arren, beso robotikoaren zehaztasuna hain handia ez zenez, batzuetan pintzaren posizioa ez zen guztiz egokia. Hori dela eta, pieza hartzerakoan, pintza ixtean, batzuetan honek kale egin eta pieza hartu gabe gelditzen zen.

Bi arazo hauek egon arren, beraien maiztasuna oso txikia zenez, errealitateko piezen manipulazioa modu egokian garatzen dela esan daiteke, eta, beraz, proiektu honek zituen helburu nagusiak betetzen direla.

10. KAPITULUA

Ondorioak eta etorkizunerako lana

Azkeneko atal honetan proiektuaren garapenean zehar ondorioztatutakoa azalduko da. Honez gain, proiektu honek etorkizunean eduki ditzakeen hobekuntzak ere agertzen dira.

10.1 Ondorioak

Proeiktuaren garapena hasi aurretik finkatutako helburu guztiak proiektua garatu ondoren lortu dira. Sistemak eduki beharreko portaera simulatu eta errealitatean ere probatu da, emaitza onak emanez. Hauek dira proiektuaren garapenean zehar ateratako ondorio nagusienak:

- ROSen garrantzia:
  Robotika munduan garrantzi handia duen *framework* hau erabiltzen ikastea etorkizunean oso erabilgarria gerta daiteke. Munduko robot gehienek darabiltzate eta mundu honetan ia ezinbesteko gaitasun bilakatu da erraminta hau erabiltzen jakitea. Proiektu honen bidez, *framework* honen garrantziaz jabetzeaz gain, berau erabiltzeko gaitasuna garatu da praktikaren bitartez.

- Prozesuen aldiberekotasuna:
  ROSekin batera aldiberean erabiltzen ziren sentsoreak eta plugin guztien koordinazioa gauzatzea lan zaila da. Momentu berean hainbat prozesuren kontrola modu eraginkor eta zuzen batean gauzatzea proiektuaren erronkarik zailenetako bat bilakatu da.

- Koordinatu sistemak:
  Proiektuaren zailtasun handi bat bilakatu ziren koordinatu sistema guztien bateragarritasuna lortzea, eta, batez ere, *tf* paketearen konfigurazioa gauzatzea. Pakete horren bidez errealitateko sistema rViz bistaratzailean modu zuzenean errepresentatzea lan konplexua dela ikusi da.

- Bistaratzaile eta simuladoreen laguntza:
  Proiektuaren garapenari esker bistaratzaile eta simulazioen laguntza ia ezinbestekoa dela ondorioztatu da. rViz bezalako bistaratzaileen laguntzaz roboten uneko egoera, sentsoreen informazioa, inguruaren errepresentazioak... ikus daitezke. Simuladoreen bidez (Moveit!-ek inplementatzen duena bezalakoa) errealitatean neketsuak, garestiak edota arriskutsuak izan daitezkeen probak egitea ahalbidetzen da. Bi erreaminta hauen bidez gure proiektuaren garapena errazagoa bilakatzen da, eta lan egiten dugun elementuak modu errazagoan ulertzea ahalbidetzen digu.

- Simulazio/errealitate dualitatea:
  Simulazioen edota bistaratzeen bidez lortzen eta ikusten duguna errealitatearekin bat beti egiten ez duela nabaritu da. Errealitatean agertzen diren eta simulazioan kontuan hartzen ez diren hainbat faktoreengatik baliteke bat ez etortzea. Simulatzaile eta bistaratzaileekin egindako lana errealitatearekin ahalik eta modu zehatzenean errepresentatzen saiatu arren, zaila gertatzen da errealitatea guztiz simulatzea.

- Ikusmean artifizialaren zailtasunak:
  Gure proiektuan erabili dugun Kinect sentsorearekin lan egiterakoan, ikusmen artifiziala aplikatzeko orduan zailtasun handia nabaritu da. Gizakiok koloreak, formak, ehundurak... identifikatzeko modua sentsore optikoek hautematen dutenaren oso desberdina da. Adibidez, gizakiok tonalidade desberdineko urdinak kolore urdinekoak izango balira identifikatu arren, sentsore optikoek kolore guztiz desberdinak izango balira bezala interpretatzen dituzte. Honez gain, sentsore mota hauekin lan egiterakoan argiztapenak duen garrantzi handia hauteman da. Argiztapenaren arabera, sentsoreek kolore bera modu desberdinetara uler dezakete, ikusmen artifizialak suposatzen dituen arazoak areagotuz.

10.2 Etorkizunerako hobekuntza posibleak

Azkeneko atal honetan etorkizun batean proiektu honi aplikatu ahal diren hobekuntza posible batzuk azalduko dira. Hobekuntza hauek, hasierako helburuetatik at egon arren,

sistemaren erabilpenak zabalduko lituzkete, edota dagoeneko inplementatuta dagoena hobetuko lukete.

- Posizioen aurredefinizioa ezabatzea:
  Proiektuan garatutako sisteman aldez aurretik definitutako hiru posizio desberdinetan jar daitezke piezak manipulatuak izateko. Hobekuntza moduan, Kinect sentsoreak emandako posizioarekin, transformatuak garatuz, beso robotikoarekiko posizioarekiko pieza non dagoen kalkulatu eta manipulatzea izango litzateke hobekuntzetako bat. Modu honetan, beso robotikoaren irismenaren barruan pieza edozein posiziotan jarrita, berau manipulatuko luke.

- manipulazio-abiadura:
  Sistemako beso robotikoaren motoreen abiadura oso motela da. Beso robotiko hau zehatzagoa eta azkarragoa den beste batekin trukatuz gero, piezen manipulazioa askoz ere azkarrago garatuko genuke.

- 3D kameraren bereizmena:
  Proiektuan erabilitako Kinect sentsorea ez da oso zehatza eta bereizmen txikikoa da (batez ere RGB kamera). Bere hurrengoa den KinectV2 sentsorea erabiliz objektuen eta oztopoen identifikazioa zehatzagoa izango litzateke.

Eranskinak

A. ERANSKINA

E1 eranskina: Mover4 beso robotikoaren elementuen definizioa (*CPRMmover4.srdf.xacro*)

```
<link name="base_link">
    <visual>
    <origin rpy="0 0 0" xyz="0 0 0.0"/>
      <geometry>
    <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint0.obj" scale="0.001 0.001
     0.001"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 0.8"/>
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint0Coll.obj" scale="0.001
     0.001 0.001"/>
      </geometry>
    </collision>
  </link>

 <link name="geoJoint1">
    <visual>
    <origin rpy="0 0 0" xyz="0 0 0.0"/>
      <geometry>
    <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint1.obj" scale="0.001 0.001
     0.001"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 0.8"/>
```

```
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint1Coll.obj" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>

<link name="geoJoint2">
    <visual>
    <origin rpy="0 0 0" xyz="0 0 0.0"/>
      <geometry>
    <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint2.obj" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 0.8"/>
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint2Coll.obj" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>

<link name="geoJoint3">
    <visual>
    <origin rpy="0 0 0" xyz="0 0 0.0"/>
      <geometry>
    <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint3.obj" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="black">
        <color rgba="0 0 0 0.8"/>
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint3Coll.obj" scale="0.001 0.001 0.001"/>
      </geometry>
    </collision>
  </link>

<link name="geoJoint4">
```

```
74      <visual>
75      <origin rpy="0 0 0" xyz="0 0 0.0"/>
76        <geometry>
77      <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint4.obj" scale="0.001 0.001
        0.001"/>
78        </geometry>
79        <material name="black">
80          <color rgba="0 0 0 0.8"/>
81        </material>
82      </visual>
83      <collision>
84        <origin xyz="0 0 0" rpy="0 0 0" />
85        <geometry>
86          <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/Joint4Coll.obj" scale="0.001
        0.001 0.001"/>
87        </geometry>
88      </collision>
89    </link>
90
91  <link name="gripperBody">
92      <visual>
93      <origin rpy="0 0 0" xyz="0 0 0.0"/>
94        <geometry>
95      <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/GripperBase.obj" scale="0.001
        0.001 0.001"/>
96        </geometry>
97        <material name="black">
98          <color rgba="0 0 0 0.8"/>
99        </material>
100     </visual>
101     <collision>
102       <origin xyz="0 0 0" rpy="0 0 0" />
103       <geometry>
104         <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/GripperBaseColl.obj" scale
        ="0.001 0.001 0.001"/>
105       </geometry>
106     </collision>
107   </link>
108
109 <link name="gripperFinger1">
110     <visual>
111     <origin rpy="0 0 0" xyz="0 0 0.0"/>
112       <geometry>
113     <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/GripperFinger.obj" scale="0.001
        0.001 0.001"/>
114       </geometry>
115       <material name="black">
116         <color rgba="0 0 0 0.8"/>
117       </material>
118     </visual>
119     <collision>
120       <origin xyz="0 0 0" rpy="0 0 0" />
```

```
121       <geometry>
122         <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/GripperFingerColl.obj" scale
      ="0.001 0.001 0.001"/>
123       </geometry>
124     </collision>
125   </link>
126
127 <link name="gripperFinger2">
128     <visual>
129     <origin rpy="0 0 0" xyz="0 0 0.0"/>
130       <geometry>
131     <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/GripperFinger.obj" scale="0.001
      0.001 0.001"/>
132       </geometry>
133       <material name="black">
134         <color rgba="0 0 0 0.8"/>
135       </material>
136     </visual>
137  <collision>
138       <origin xyz="0 0 0" rpy="0 0 0" />
139       <geometry>
140         <mesh filename="package://cpr_rviz_plugin/GeometryCPRMover4/GripperFingerColl.obj" scale
      ="0.001 0.001 0.001"/>
141       </geometry>
142     </collision>
143   </link>
```

B. ERANSKINA

E2 eranskina: Mover4 beso robotikoaren loturen definizioa

```
<joint name="Joint0" type="revolute">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="geoJoint1"/>
    <origin rpy="0 0 0" xyz="0 0 0.158"/>
    <limit effort="100" lower="-1" upper="2" velocity="30" />
    <joint_properties damping="0.0" friction="0.0"/>
</joint>

<joint name="Joint1" type="revolute">
    <axis xyz="0 1 0"/>
    <parent link="geoJoint1"/>
    <child link="geoJoint2"/>
    <origin rpy="0 0 0" xyz="0 0 0.060"/>
    <limit effort="100" lower="-0.5" upper="1" velocity="100"/>
    <joint_properties damping="0.0" friction="0.0"/>
</joint>

<joint name="Joint2" type="revolute">
    <axis xyz="0 1 0"/>
    <parent link="geoJoint2"/>
    <child link="geoJoint3"/>
    <origin rpy="0 0 0" xyz="0 0 0.190"/>
    <limit effort="100" lower="-0.6" upper="2.4" velocity="100"/>
    <joint_properties damping="0.0" friction="0.0"/>
</joint>

<joint name="Joint3" type="revolute">
    <axis xyz="0 1 0"/>
```

```
    <parent link="geoJoint3"/>
    <child link="geoJoint4"/>
    <origin rpy="0 0 0" xyz="0 0 0.220"/>
    <limit effort="100" lower="-2.2" upper="2.2" velocity="100"/>
    <joint_properties damping="0.0" friction="0.0"/>
</joint>

<joint name="GripperFixed" type="fixed">
    <parent link="geoJoint4"/>
    <child link="gripperBody"/>
    <origin rpy="0 0 0" xyz="0 0 0.046"/>
</joint>

<joint name="Gripper1" type="revolute">
    <axis xyz="1 0 0"/>
    <parent link="gripperBody"/>
    <child link="gripperFinger1"/>
    <origin rpy="0 0 3.141" xyz="0.0 0.01 0.03"/>
    <limit effort="100" lower="0" upper="0.62" velocity="100"/>
    <joint_properties damping="0.0" friction="0.0"/>
</joint>

<joint name="Gripper2" type="revolute">
    <axis xyz="1 0 0"/>
    <parent link="gripperBody"/>
    <child link="gripperFinger2"/>
     <origin rpy="0 0 0" xyz="0.0 -0.01 0.03"/>
    <limit effort="100" lower="0" upper="0.62" velocity="100"/>
    <joint_properties damping="0.0" friction="0.0"/>
</joint>
```

C. ERANSKINA

E3 eranskina: Mover4 beso robotikoaren manipulazio-begizta *move_group* interfazea erabiliz

```
1
2  //move_group interfazearekin lan egiteko beharrezko dependentziak
3
4  #include <ros/ros.h>
5  #include <iostream>
6  #include <string.h>
7  #include <actionlib/client/simple_action_client.h>
8  #include <actionlib/client/terminal_state.h>
9  #include <control_msgs/GripperCommandAction.h>
10 #include <moveit/move_group_interface/move_group.h>
11 #include <std_msgs/String.h>
12 #include <sensor_msgs/JointState.h>
13 #include <geometry_msgs/Twist.h>
14 #include <moveit/planning_scene_interface/planning_scene_interface.h>
15
16 //aldagaien erazagutzea
17
18 float jointPos[6];
19 int i;
20 bool same;
21 std::string objectPosition;
22
23   ros::Publisher gripperCommand;
24   ros::Subscriber states;
25   ros::Subscriber objPos;
26
27 std_msgs::String command;
28 std::vector<double> group1_variable_values;
29 std::vector<double> group1_actual_values;
```

```
30
31  moveit::planning_interface::MoveGroup::Plan my_plan;
32  bool success, firstTime;
33
34  //robot erreala mugitzeko errutina
35
36  void jointStateCallback1(const sensor_msgs::JointState msg){
37      jointPos[0]=msg.position[0];
38      jointPos[1]=msg.position[1];
39      jointPos[2]=msg.position[2];
40      jointPos[3]=msg.position[3];
41  }
42
43  //nodoen arteko mezuak jasotzeko errutina
44
45  void objPosCallback(const std_msgs::String::ConstPtr& msg){
46          objectPosition = msg->data.c_str();
47  }
48
49  //Programa nagusia
50
51  int main(int argc, char **argv)
52  {
53    ros::init(argc, argv, "proba");
54
55    ros::NodeHandle nh_;
56
57
58    ros::AsyncSpinner spinner(1);
59    spinner.start();
60
61    firstTime=true;
62
63    //Pintzei eragiteko topic-ean idazteko aldagaia (publisher-a)
64    gripperCommand =  nh_.advertise<std_msgs::String>("CPRMoverCommands", 1, true);
65
66    //Robotaren egoera jasotzeko aldagaia (subscriber-a)
67    states = nh_.subscribe<sensor_msgs::JointState>("/joint_states", 1,jointStateCallback1 );
68
69    //Objektuen posizioa jasotzeko aldagaia (subscriber-a)
70    objPos = nh_.subscribe<std_msgs::String>("/segment_camera_image/position", 1,objPosCallback );
71
72    //"robot" ardatz-taldearentzat kalkulatu mugimenduak
73    moveit::planning_interface::MoveGroup group1("robot");
74
75    //Pintza ireki
76    command.data = "GripperOpen";
77    gripperCommand.publish(command);
78
79    group1.setPoseReferenceFrame ("base_link");
80    ROS_INFO("Reference frame: %s", group1.getPlanningFrame().c_str());
81
```

```
82
83    moveit::planning_interface::PlanningSceneInterface planning_scene_interface;
84    sleep(5);
85
86    while(true){
87
88      group1.getCurrentState()->copyJointGroupPositions(group1.getCurrentState()->getRobotModel()->
       getJointModelGroup(group1.getName()),  group1_variable_values);
89
90      firstTime=true;
91      printf("%s\n",objectPosition.c_str());
92
93      //Pieza lehenengo posizioan badago, posizio horretara joan
94      if(objectPosition == "one"){
95          group1_variable_values[0] = -0.7416;
96          group1_variable_values[1] = 1.0365;
97          group1_variable_values[2] = 1.1429;
98          group1_variable_values[3] = 0.7846;
99      //Pieza bigarrengo posizioan badago, posizio horretara joan
100     }else if(objectPosition == "two"){
101         group1_variable_values[0] = -0.00879;
102         group1_variable_values[1] = 1.0365;
103         group1_variable_values[2] = 1.1429;
104         group1_variable_values[3] = 0.7846;
105     //Pieza hirugarren posizioan badago, posizio horretara joan
106     }else{
107         group1_variable_values[0] = 0.5409;
108         group1_variable_values[1] = 1.0365;
109         group1_variable_values[2] = 1.1429;
110         group1_variable_values[3] = 0.7846;
111     }
112
113     group1.setJointValueTarget(group1_variable_values);
114     success = group1.plan(my_plan);
115
116     //Mugimendua posiblea bada inongo objekturekin talka egin gabe, robota mugitu
117     group1.move();
118
119     bool same=false;
120
121     while(same == false){
122         group1.getCurrentState()->copyJointGroupPositions(group1.getCurrentState()->getRobotModel
       ()->getJointModelGroup(group1.getName()), group1_actual_values);
123
124         for(i=0;i<4;i++){
125             if(group1_actual_values[i] < group1_variable_values[i]){
126                 same=false;
127             }else{
128                 same=true;
129             }
130         }
131     }
```

```
132
133     //Pintza itxi
134     command.data = "GripperClose";
135     gripperCommand.publish(command);
136
137     //Piezak uzteko posiziora mugitu robota
138     group1_variable_values[0] = 1.631833;
139     group1_variable_values[1] = 1.0365;
140     group1_variable_values[2] = 1.1429;
141     group1_variable_values[3] = 0.7846;
142     group1.setJointValueTarget(group1_variable_values);
143
144     //Mugimendua posiblea bada inongo objekturekin talka egin gabe, robota mugitu
145     success = group1.plan(my_plan);
146
147     group1.move();
148
149     same=false;
150
151     while(same == false){
152        group1.getCurrentState()->copyJointGroupPositions(group1.getCurrentState()->getRobotModel()
       ->getJointModelGroup(group1.getName()),        group1_actual_values);
153
154        if(group1_actual_values[0] > group1_variable_values[0] && group1_actual_values[1] <
       group1_variable_values[1] ){
155           same=true;
156        }
157     }
158
159     //Pintza ireki eta pieza utzi
160     command.data = "GripperOpen";
161     gripperCommand.publish(command);
162
163     //Robota atseden posiziora mugitu
164     group1_variable_values[0] = 0;
165     group1_variable_values[1] = -0.2824;
166     group1_variable_values[2] = 1.7649;
167     group1_variable_values[3] = -0.3177;
168     group1.setJointValueTarget(group1_variable_values);
169
170     success = group1.plan(my_plan);
171
172     group1.move();
173
174     sleep(15);
175  }
176 }
```

Bibliografia

[1] Robot Operating System (ROS). http://ros.org. [2017ko maiatzean atzitua]

[2] rViz. http://wiki.ros.org/rviz. [2017ko apirilean atzitua]

[3] Moveit!. http://moveit.ros.org/. [2017ko maiatzean atzitua]

[4] Microsoft Kinect 3D kamera. https://msdn.microsoft.com/en-us/library/jj131033.aspx. [2017ko maiatzean atzitua]

[5] rViz tutorialak. wiki.ros.org/rviz/Tutorials. [2017ko maiatzean atzitua]

[6] Moveit! tutorialak. http://docs.ros.org/indigo/api/moveit_tutorials/html/index.html. [2017ko maiatzean atzitua]