Treball de Fi de Grau

Grau en Tecnologies Industrials

ALGORISMES D'INTERACCIÓ AMB L'ENTORN PER AL ROBOT DARWINOP

MEMÒRIA

Autor: Júlia Marsal Perendreu
Director/s: Dr. Guillem Alenyà Ribas
Sr. Sergi Hernández Juan

Convocatòria: Setembre 2016

ETSEIB

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

UPC

Capítol 1

RESUM

El present projecte és el resultat del desenvolupament i la implementació d'un algorisme d'interacció amb l'entorn utilitzant un nou programari base per al robot DarwinOP.

El treball es basa en la realització de la prova de visió d'un concurs, la qual utilitza la càmera i una llibreria de detecció de codis QR per poder detectar i descodificar el codi QR que té al davant el robot i, girar tants graus cap a la direcció on digui fins trobar el següent codi QR. I així successivament, fins descodificar els 8 codis QR de la prova.

Per realitzar les proves amb el robot i evitar la seva deterioració per l'ús, s'utilitza el simulador *gazebo* durant el desenvolupament de la prova. Després amb l'algoritme ja creat, s'utilitza el robot real.

El projecte inclou en la primera part una descripció de les eines de *hardware* i de *software* utilitzades en el robot DarwinOP. Així com la descripció dels components del robot, les llibreries, l'entorn *ROS* i la descripció del simulador *Gazebo* 4.1.2. Per altra banda, la segona part inclou la realització i implementació de la prova de visió [5].

El desenvolupament d'aquest projecte s'ha fet de forma modular, de tal manera que es pugui anar expandint i optimitzant l'algoritme realitzat de forma que el procés cada cop sigui més robust i tingui en compte una casuística més completa.

Índex

Índex de figures

Part I

PART I

Capítol 2

PREFACI

2.1 Origen del projecte

El fet de que l'autora participés en el concurs de robòtica hominoide CEABOT l'any 2015, organitzat pel *comitè d'automàtica* en el congrés d'automàtica de Bilbao [4], amb un robot hominoide *BIOLOID*[35] programat en el llenguatge de programació C va despertar l'interès de l'autora cap a l'aprenentatge de nous llenguatges i estructures robòtiques que permetin fer algoritmes més complexos i efectius.

Per això, i gràcies a l'aprovació del laboratori de *percepció i manipulació*, l'estància a l'iri *l'IRI*[32] hel projecte final de grau amb el robot hominoide *Darwin*.

2.2 Motivació

Ja des des de batxillerat, l'autora ha tingut molta inquietud sobre aquest món tan gran i meravellós de de la robòtica. Ha anat realitzant diferents projectes durant la carrera, com ara robots simples seguidors de línies, una plataforma *Stewart*, i tallers de robòtica educativa a nens i adolescents, però a un nivell molt bàsic. Va ser el concurs de l'any *2015*, esmentat en l'apartat anterior, el que li va donar la màxima motivació per a entrar al món de la robòtica utilitzada a nivell industrial i en la investigació.

2.3 Requeriments previs

El projecte requereix coneixements previs de programació en *C++* [27], del *framework ROS* [18] per fer funcionar el robot i, el llenguatge de creació de textos *latex*, per escriure la memòria.

Capítol 3

INRODUCCIÓ

El robot humanoide darwinOP conté un programari bàsic que serveix per realitzar tasques bàsiques, com ara caminar i fer gestos. Però, es pot completar aquest programari per a fer tasques més complexes? La resposta es troba en el projecte actual, plantejant-se el comportament tant en simulació com en el robot real.

3.1 OBJECTIUS DEL PROJECTE

L'objectiu general del projecte és que el robot humanoide darwinOP realitzi de forma autònoma, la prova de visió d'un concurs utilitzant un programari bàsic més complex que el que utilitzava fins ara.

Per complir l'objectiu general s'ha d'estudiar la viabilitat dels següents objectius específics:

- Estudiar el funcionament del robot *darwinOP*, tant a nivell de *software* com de *hardware*.
- Estudiar el funcionament de la càmera i de les llibreries necessàries per obtenir la posició del centre del codi QR i posicionar la càmera centrada amb el codi QR.
- Desenvolupar un algoritme que realitzi la prova del concurs utilitzant el robot *DarwinOP*, la càmera i la llibreria de detecció dels codis QRs.
- Desenvolupar l'algoritme mitjançant l'eina de la simulació.
- Validar l'algoritme al robot real comparant-lo amb els resultats simulats.
- Crear un nou suport que permeti que la càmera s'aguanti de forma paral·lela als codis QRs per detectar-los més fàcilment i nítidament.

- Documentar tot el *software* i *hardware* utilitzat en la nova infraestructura del robot per al seu ús futur.

3.2 ABAST DEL PROJECTE

L'abast del present projecte inclou el disseny de l'algoritme per realitzar la prova de visió del concurs CEABOT, utilitzant un programari més complex al que s'havia utilitzat fins ara.

El projecte no inclou la modificació dels components del robot actual. Però si la incorporació d'una càmera diferent a la de sèrie i un suport per aguantar-la.

Un cop obtingut l'algoritme, es provarà mitjançant un simulador i amb el mateix robot real.

Capítol 4

CONEIXEMENTS PREVIS

El fet de realitzar el projecte a l'IRI [32] utilitzant el robot DarwinOP [7], ha condicionat els coneixements previs a adquirir.

Primerament, en l'apartat 4.1 es parlarà de l'estructura de treball de l'IRI, així com, quines eines de desenvolupament utilitza, com s'utilitzen i quina utilitat tenen. En l'apartat 4.1.1 es parla de l'estructura de treball, en l'apartat 4.1.2 s'explica què és ROS, en l'apartat 4.1.2 es parla del simulador què s'utilitza i finalment, en l'apartat 4.1.3, es parla d'una estructura de treball què l'IRI utilitza per fer més senzilla la forma de treball.

Seguidament, en l'apartat 4.2 es parla del robot DarwinOP [7]. Aquest conté una breu descripció de les parts i característiques del robot en l'apartat 4.2.1, una descripció de les característiques de la càmera en l'apartat 4.2.3 i, una descripció de l'estructura del nou *firmware* desenvolupat per l'IRI en l'apartat 4.2.4.

4.1 Estructura amb què es treballa a l'IRI:

A l'IRI (Institut de Robòtica Industrial) [32] es treballa amb un entorn de desenvolupament propi, el qual es defineix a *Labrobotica* 4.1.1. L'estructura general d'aquest marc de treball es pot veure a la Fig. 4.1.

Les eines de desenvolupament que generalment s'utilitzen a l'IRI són:

- **C/C++**: aquest és el llenguatge de programació més habitual per desenvolupar tant *drivers* de *Linux* per accedir a dispositius *hardware* com podrien ser sensors o robots, i també algoritmes d'alt nivell que implementin noves funcionalitats de navegació, percepció, manipulació, etc. per als robots. A la secció 4.1.1 es descriu amb més detall l'estructura utilitzada a l'IRI.

- **ROS**: Aquest és un marc de treball d'ús molt generalitzat en el món de la robòtica el qual simplifica el desenvolupament de sistemes robòtics complexos formats per múltiples mòduls funcionant en diversos ordinadors connectats en xarxa. El *middleware* [16] ROS ofereix una abstracció de les comunicacions entre mòduls de *software* que facilita tornar a utilitzar codi propi i també de codi extern a l'Institut. A més a més també ofereix un conjunt d'eines de visualització, simulació, test i diagnòstic que disminueixen considerablement el temps de desenvolupament. A la secció 4.1.2 s'introdueix en detall l'entorn de treball ROS, i a la secció 4.1.3 s'explica l'estructura particular que s'utilitza a l'IRI, que se li ha donat el nom de IRI-ROS.

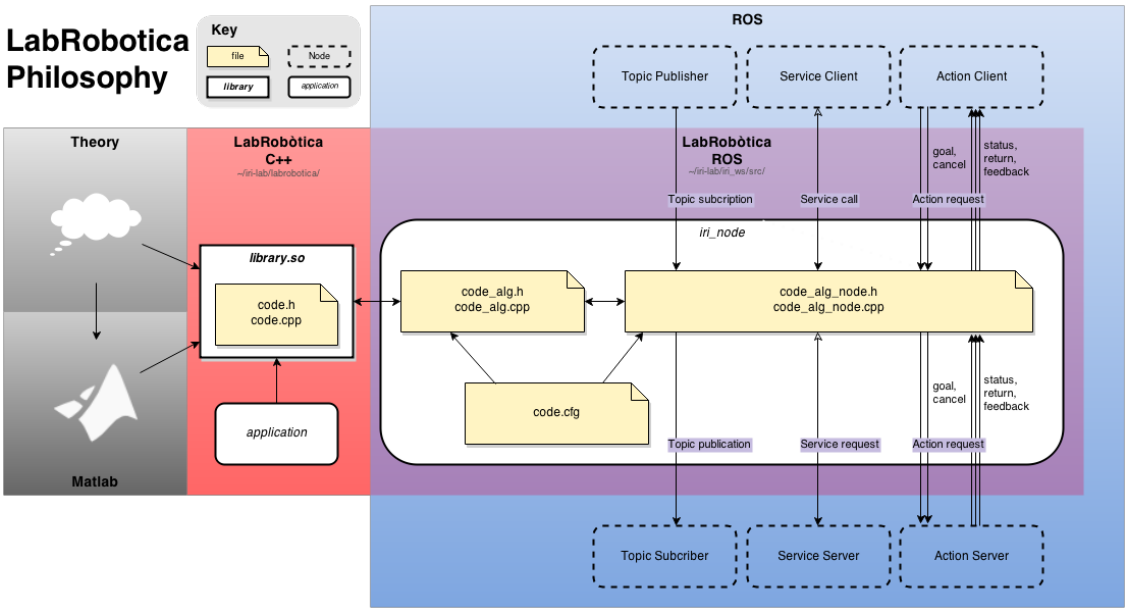


Figura 4.1: Estructura de treball a l'iri.

El fet de separar la implementació de les funcionalitats desitjades (ja siguin drivers de dispositius físics o algoritmes) de la part de comunicació entre els diferents mòduls del sistema complet (ROS), facilita la depuració d'errors i també la col·laboració amb entitats externes que no tenen per que utilitzar ROS.

Finalment, per tal de facilitar el manteniment i gestió del software desenvolupat, es treballa amb el gestor de repositoris subversion o SVN [30].

4.1.1 Labrobotica

A nivell de C/C++, com es pot veure a la part acolorida en rosa de la Fig. 4.1, es treballa bàsicament amb llibreries dinàmiques (.so en sistemes Linux). Aquestes llibreries estan pensades

per encapsular tota la funcionalitat desitjada per tal que es pugui utilitzar després a la capa ROS, o bé en qualsevol altre entorn.

Per facilitar tant el manteniment del codi com la seva re-utilització dins de l'Institut s'ha proposat una estructura comuna, tant a nivell d'estructura de directoris dels projectes de desenvolupament, com de les eines utilitzades.

La informació més detallada de l'estructura de directoris proposada es pot trobar a [9]. Per facilitar l'ús d'aquesta estructura comuna hi ha un script disponible que permet crear una plantilla bàsica a partir de la qual començar a desenvolupar de forma ràpida. Aquest script està descrit amb detall a la següent pàgina de la wiki de l'Institut [11].

Pel que fa a les eines de desenvolupament utilitzades, cal destacar:

- El compilador estàndard de Linux (gcc).
- L'eina cmake per simplificar la creació de scripts de compilació Makefile.
- Doxygen per generar la documentació que es penjarà a Internet quan el software es faci públic.
- Degut a la gran diversitat d'entorns de desenvolupament integrats (IDE), cada persona pot fer servir el que li resulti més còmode, però no n'hi ha cap de recomanat.

4.1.2 ROS

El sistema operatiu de robòtica modular; *Robotics Operating System*, és una infraestructura de treball que permet fer un desenvolupament de forma estructurada, útil i, efectiva, facilitant la programació i interconnexió entre les parts que el robot conté. [18].

Es tracta d'un sistema què és divideix en conjunt de mòduls(nodes) que poden intercanviar informació a través de missatges. També, conté eines i llibreries que ajuden a obtenir, construir, escriure i executar codi entre diferents ordinadors.

L'estructura bàsica de ros es representa a la figura 4.2. A continuació s'explicaran les diferents estructures que s'utilitzen:

Nodes:

Els nodes [20] són bàsicament executables què utilitzen eines de ROS per a comunicar-se amb altres nodes mitjançant els tòpics, serveis i accions.

Els avantatges d'utilitzar nodes són els següents:

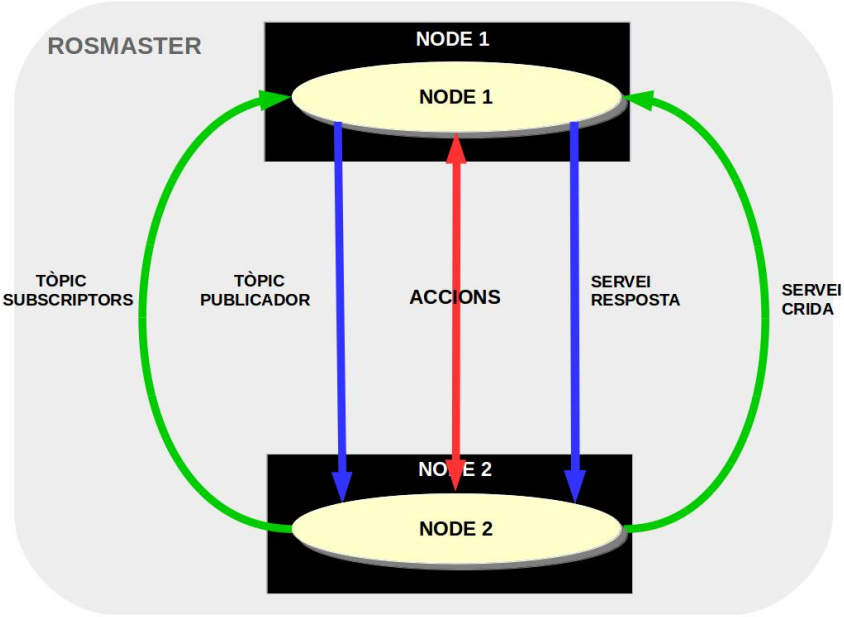


Figura 4.2: Estructura bàsica de ROS.

- Permet aïllar funcionalitats que estan dins d'un mateix context.
- La complexitat dels codis es redueix, ja que cada node tindrà el seu propi codi.
- Permet tenir diferents algoritmes i executar-los en cas de que siguin necessaris de forma senzilla, en el mateix *workspace*.

Paquets:

Els nodes es creen dins de *paquets*, que contenen els fitxers necessaris per definir el node. En la figura 4.2 existeixen dos nodes (en *groc*) dins de *paquets* (en *negre*).

Tòpics:

Els tòpics[22] són canals de comunicació unidireccional que permeten enviar informació d'un node (publicador) a un o més nodes (subscriptors) a una freqüència donada. No té cap tipus de verificació sobre si la informació s'ha rebut correctament o no.

La informació bàsica que s'envia és un missatge (*msg*). A la figura 4.2 s'observa com el node 1 conté un **subscriptor** del node2 (línia verda) i, és un **publicador** del node2.

En els nodes, existeixen funcions anomenades *callbacks*,les quals reben la informació del tòpic on aquest esta subscrit.

Serveis:

Els serveis [**?**] formen part d'una altra forma d'intercanvi d'informació. Els serveis funcionen en forma de **crida/resposta**, tal com es representa en la imatge 4.3. Entre dos nodes o parts d'un robot, un cridarà a l'altre quan únicament necessiti aquella informació i, l'altre enviarà la informació únicament quan el cridin.

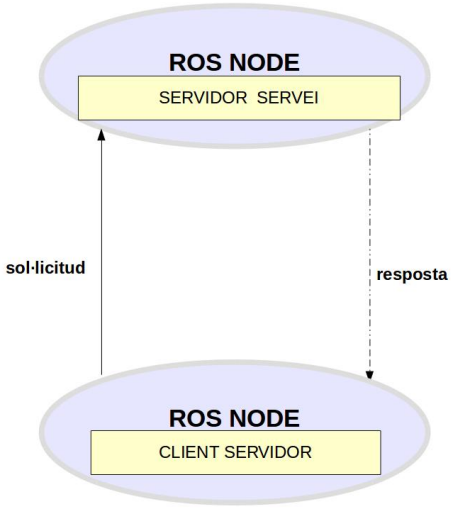


Figura 4.3: Funcionament del servei

En l'imatge 4.2 la línia verda és la que fa referència a la sol·licitud i, la blava, és la que fa referència a la resposta.

Accions:

Les accions[17] apareixen per poder fer accions sota demanda com en el cas dels serveis, però l'execució d'aquestes accions es pot allargar en el temps. Com que no es viable mantenir el node sol·licitant bloquejat fins que l'acció acabi, aquest tipus de comunicació retorna immediatament quan s'ha enviat la sol·licitud (*goal*) i permet saber de la seva evolució a través de missatges de *Feedback* i d'estat (*result*). També és possible cancel·lar una acció activa, ja sigui cancel·lant-la o bé enviant un altre goal.

Els estats d'una acció poden ser *PREEMPTED* o cancel·lat per causes externes, *ABORTED* o cancel·lat per problemes interns, *ACTIVE* quan l'acció s'està executant, i, quan l'acció acaba correctament *SUCCEEDED*.

En l'imatge 4.2 l'acció serà el de color *vermell*, ja que permet l'enviament d'informació i l'obtenció de l'estat de l'acció.

Master:

El *Master*[19] és el *cervell*, què controla totes les diferents interconnexions (*tòpics, serveis i accions*) entre els diferents *nodes*.

En el cas de la figura 4.2 és l'encarregat de controlar la interconnexió entre els diferents tòpics, serveis i accions dels dos nodes.

Servidor de paràmetres:

El servidor de paràmetres[21] forma part del *Master*, que permet guardar dades mitjançant una clau en una localització central.

Namespaces:

Els **Noms** tenen un paper molt important en ROS: els nodes, tòpics, serveis i paràmetres tenen un nom. Cada llibreria de ROS suporta un **remapping**, què vol dir que un programa compilat és torna a configurar durant l'execució per a ajustar-se a la topologia i assignació de noms dels nodes de cada sistema ROS.

Simulador Gazebo:

Per realitzar la primera part de l'experimentació s'utilitza el simulador *Gazebo* [29].

Els **avantatges** d'utilitzar un simulador són els següents:

- Facilita la detecció i la correcció dels errors.
- Valida que l'algoritme funciona bé, eliminant la problemàtica associada a l'ús del robot real.
- Redueix el cost del desgast del *hardware* del robot real.
- Permet més flexibilitat a l'hora de triar i modificar l'entorn on es mou el robot.

Els **desavantatges** d'utilitzar un simulador són els següents:

- En l'execució amb el robot real, el comportament depèn de molts elements del *hardware*, fet que el comportament del robot pugui variar respecte la simulació.

Per tant, el simulador que s'utilitza amb el robot darwinOP i, amb l'entorn adaptat per prova del CEABOT[4], tal com es mostra en la figura 4.4.

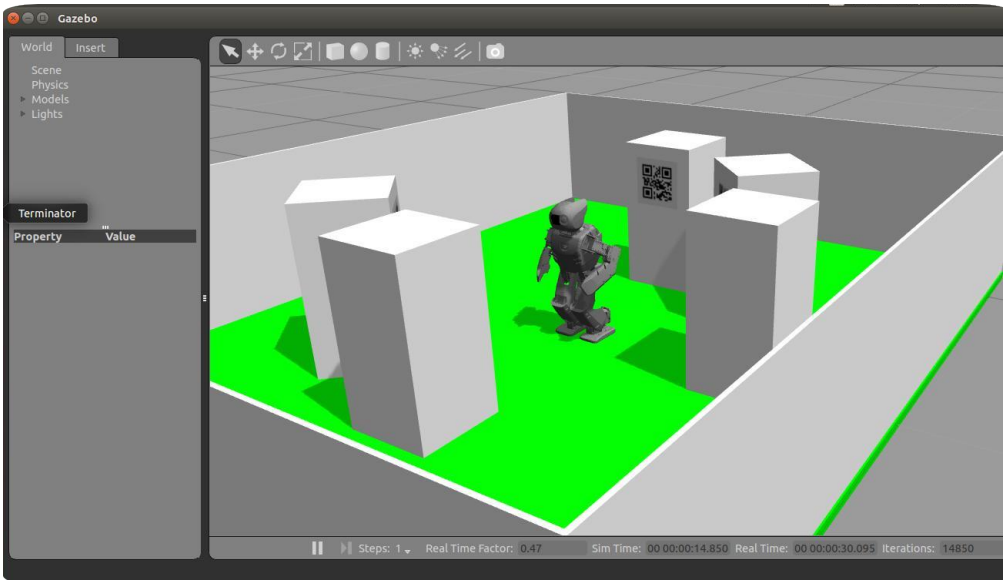


Figura 4.4: Representació amb la simulació

4.1.3 IRI-ROS

Tal i com s'ha fet a nivell de C/C++, a nivell de ROS també s'ha proposat des de l'IRI[32] una estructura comuna de treball que simplifiqui el desenvolupament en l'entorn ROS i que faciliti la re-utilització del codi i el seu manteniment. Aquesta estructura es coneix com a IRI-ROS i la seva estructura general es pot veure a la part acolorida en blau de la Fig. 4.1

Aquesta estructura es divideix en dues parts ben diferenciades, ambdues implementades com a objectes de C++:

- **Interfície amb el driver o algoritme de C/C++:** (code_alg a la Fig. 4.1) Aquesta part té com a objectiu principal interaccionar directament amb la llibreria de C/C++, oferint una interfície (API) simplificada al nivell del node. En el cas particular del *driver* d'un dispositiu físic, aquesta part implementa la màquina d'estats que es pot veure a la Fig. 4.5 per tal de gestionar la correcta configuració i posada en marxa del dispositiu. Al iniciar el node de ROS, les funcions **doOpen()** i **doStart()** es criden de forma periòdica en aquest ordre fins que les dues acaben amb èxit i el node passa a l'estat *RUNNING*. De la mateixa manera les funcions **doStop()** i **doClose()** es criden abans de finalitzar l'execució del node per assegurar que tots els recursos utilitzats s'han alliberat correctament. És només en l'estat **RUNNING** on el node funciona normalment i pot interactuar amb la resta de nodes de la xarxa ROS. Aquesta estructura de màquina d'estat no s'ha dissenyat i implementat al IRI, si no que s'utilitza una estructura definida a ROS anomenada driver_base que s'explica amb detall en aquesta pàgina

web [31] En el cas d'un *algoritme*, la màquina d'estats de la Fig. **??** no té cap sentit i no s'utilitza.

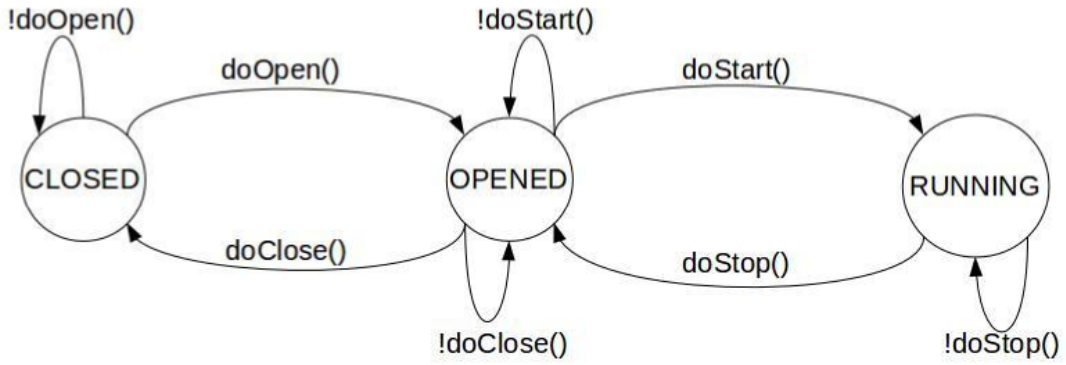


Figura 4.5: Diagrama de la màquina d'estats de control d'un node de tipus Driver

- **Interfície amb ROS per comunicar-se amb altres mòduls:** *(code_alg_node a la Fig. 4.1)* Aquesta part utilitza la API proporcionada per la primera per enviar i rebre informació cap a/des de la llibreria de C/C++, traduint entre les estructures de dades utilitzades en els missatges de ROS i les estructures de dades utilitzades per la llibreria. En aquesta part és on es defineixen totes les funcions de *callback de subscribers*, *servidors de serveis* i *servidors d'accions*. En el mode de funcionament què s'utilitza de ROS a l'Institut, totes aquestes funcions s'executen de forma seqüencial dins d'un únic fil d'execució (*thread*) quan sigui necessari. Això, en general obligaria a desenvolupar algoritmes distribuïts de control que s'executessin en diferents funcions de *callback* [3], la qual cosa augmentaria considerablement la complexitat del desenvolupament. Per aquest motiu, aquesta part també proporciona un fil d'execució paral·lel, implementat a través de la funció *MainNodeThread*, que permet treballar com si es tractés de la funció main() habitual de C/C++.

  A part dels dos nivells de codi explicats fins ara, l'estructura proposada a l'IRI també inclou les següents eines de ROS:

- **Dynamic Reconfigure**: permet definir un conjunt de paràmetres associats al node utilitzant un fitxer de configuració (code.cfg a la Fig. 4.1). Els paràmetres definits en aquest fitxer es poden modificar de forma directa a través del servidor de paràmetres de ROS (secció 4.1.2), però també es possible modificar-los a través d'una interfície gràfica simple generada de forma

automàtica. En el cas de nodes de tipus driver, tots els paràmetres han d'estar associats al primer dels nivells introduïts anteriorment degut a la pròpia estructura del driver_base. No passa el mateix amb els nodes de tipus algoritme, on els paràmetres definits en el fitxer de configuració poden estar associats a qualsevol de les dues parts.

- **Diagnòstics**: Aquesta eina de ROS permet afegir funcions que controlin qualsevol conjunt de paràmetres del driver o algoritme, així com paràmetres dels propis missatges de ROS com podria ser la seva freqüència de publicació, i genera notificacions a través d'un únic tòpic que es poden gestionar de forma centralitzada des d'una aplicació de ROS. Les notificacions poden tenir tres nivells diferents en funció de la seva severitat, que són OK, WARNING i ERROR.

- **Tests**: Aquesta eina de ROS permet afegir codi per verificar el correcte funcionament del node utilitzant la eina *gtest()* [28] de google. Degut a la complexitat inherent de l'entorn de treball que s'ha presentat fins al moment, depurar possibles errors de codi pot ser molt complicat, i aquesta eina permet reduir considerablement el temps necessari per trobar els errors i corregir-los.

  Aquesta estructura és relativament complexa per generar-la de zero cada cop que es crea un nou node de ROS. Per aquest motiu a continuació es proporcionen dos scripts que permeten la creació de forma senzilla de nodes tipus driver i de nodes tipus algoritme amb tota l'estructura i funcionalitats que s'han presentat fins ara.

  – **create_driver_package.sh**

    Aquest script genera un node bàsic de ROS de tipus driver. Els paràmetres necessaris són els següents:

    * **-n**: nom del node de ROS
    * **-i**: quan s'especifica aquest paràmetre no s'afegeix el prefix iri_ al node creat, cosa que es fa per defecte en cas contrari. Si la generació del codi acaba correctament, el nou node es compila per primera vegada de forma automàtica. Més informació sobre aquest script i com utilitzar-lo es pot trobar a [24].

  – **create_algorithm_package.sh** Aquest script genera un node bàsic de ROS de tipus algoritme. Els paràmetres necessaris són els següents:

    * **-n**: nom del node de ROS

    * **-i**: quan s'especifica aquest paràmetre no s'afegeix el prefix iri_ al node creat, cosa que es fa per defecte en cas contrari. Si la generació del codi acaba correctament, el nou node es compila per primera vegada de forma automàtica. Més informació sobre aquest script i com utilitzar-lo es pot trobar a [24].

També, afegir *publishers* o *subscribers* de tòpics, i clients o servidors de serveis o accions són tasques molt repetitives que es poden automatitzar de forma senzilla. Els scripts presentats a continuació tanmateix, permeten afegir, a nodes generats amb els *scripts* de l'IRI, *publishers* o *subscribers* de tòpics, clients o servidors de serveis i clients o servidors d'accions respectivament.

- **add_publisher_subscriber.sh** Aquest script afegeix les funcions i variables necessàries per tal de crear un publisher o un subscriber d'un tòpic qualsevol (veure secció 4.1.2). Els paràmetres necessaris són els següents:
    * **-o**: especifica si s'ha de crear un publisher o un subscriber. Els possibles valors d'aquests paràmeters són publisher i subscriber. Qualsevol altre valor generarà un error.
    * **-p**: nom del node de ROS al qual es vol afegir el publisher o subscriber. Aquest node ha d'existir dins de l'espai de treball de ROS, si no es generarà un error.
    * **-t**: nom del tòpic associat al publisher o subscriber. Al nom que s'especifica aquí s'hi afegeix el *namespace* del propi node.
    * **-m**: nom del fitxer .msg on es defineix l'estructura de dades del tòpic. El tipus de missatge ha d'existir, si no es generarà un error.
    * **-b**: mida de la cua de missatges que s'han d'emmagatzemar per a ser processats. Si la generació del codi acaba correctament, el nou node es compila per primera vegada de forma automàtica. Més informació sobre aquest script i com utilitzar-lo es pot trobar a [23].
- **add_server_client.sh** Aquest script afegeix les funcions i variables necessàries per tal de crear un client o un servidor d'un servei qualsevol (veure secció 4.1.2). Els paràmetres necessaris són els següents:
    * **-o**: especifica si s'ha de crear un client o un servidor. Els possibles valors d'aquests paràmetres són client i server. Qualsevol altre valor generarà un error.

    * **-p**: nom del node de ROS al qual es vol afegir el client o servidor. Aquest node ha d'existir dins de l'espai de treball de ROS, si no es generarà un error.
    * **-t**: nom del servei associat al client o servidor. Al nom que s'especifica aquí s'hi afegeix el namespace del propi node.
    * **-m**: nom del fitxer .srv on es defineix l'estructura de dades del servei. El tipus de missatge ha d'existir, si no es generarà un error. Si la generació del codi acaba correctament, el nou node es compila per primera vegada de forma automàtica. Més informació sobre aquest script i com utilitzar-lo es pot trobar a [2].

– **add_action_server_client.sh**[1] Aquest script afegeix les funcions i variables necessàries per tal de crear un client o un servidor d'una acció qualsevol (veure secció 4.1.2). Els paràmetres necessaris són els següents:

    * **-o**: especifica si s'ha de crear un client o un servidor. Els possibles valors d'aquests paràmetres són client i server. Qualsevol altre valor generarà un error.
    * **-p**: nom del node de ROS al qual es vol afegir el client o servidor. Aquest node ha d'existir dins de l'espai de treball de ROS, si no es generarà un error.
    * **-t**: nom del servei associat al client o servidor. Al nom que s'especifica aquí s'hi afegeix el namespace del propi node.
    * **-m**: nom del fitxer *.action* on es defineix l'estructura de dades de l'acció. El tipus de missatge ha d'existir, si no es generarà un error. Si la generació del codi acaba correctament, el nou node es compila per primera vegada de forma automàtica. Més informació sobre aquest *script* i com utilitzar-lo es pot trobar a [1].

4.2 Robot DarwinOP

El **DarwinOP** (*Dynamic Anthropomorphic Robot with Intelligence-Open Platform*) és un robot humanoide petit amb una estructura formada de sensors i actuadors sofisticats, els quals doten al robot de la possibilitat de realitzar qualsevol activitat en diferents àmbits.

4.2.1 Descripció del robot

El robot darwinOP [7] utilitzat per a realitzar la prova el que es mostra en les figures 4.7 (vista frontal), i 4.8 (vista posterior). És el mateix darwinOP que ve de fàbrica [7] però, amb el temps s'hi han afegit millores a nivell mecànic, electrònic i de *software*. Les interconnexions entre les parts

més importants es mostren en la figura 4.6.

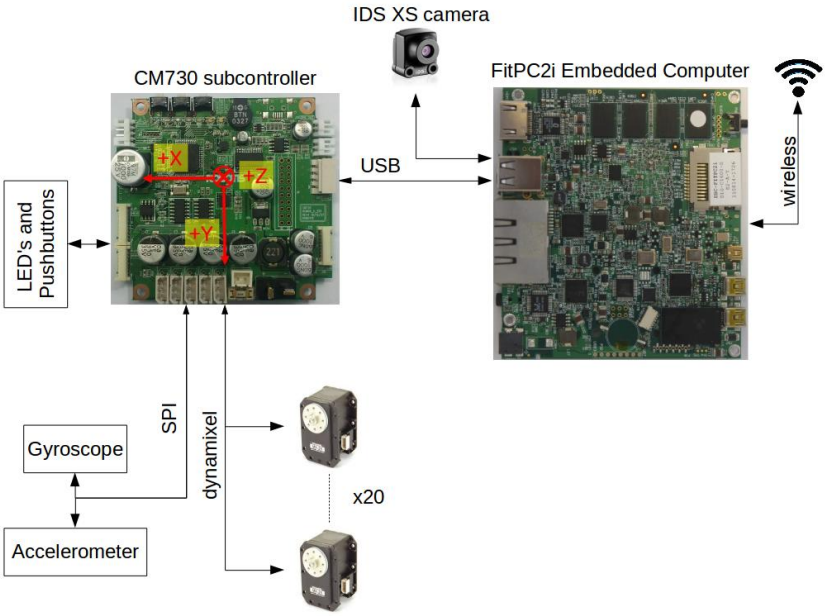


Figura 4.6: Representació de les interconnexions entre les parts més importants del robot darwinOP

Tal com mostra la figura 4.6, la controladora CM 730 [5] és la que controla tant els sensors *giroscopi i acceleròmetre* com els *servomotors dynamixels* [8]. El PC2i [10] es connecta amb la controladora mitjançant un cable USB i també a la càmera ids [14].

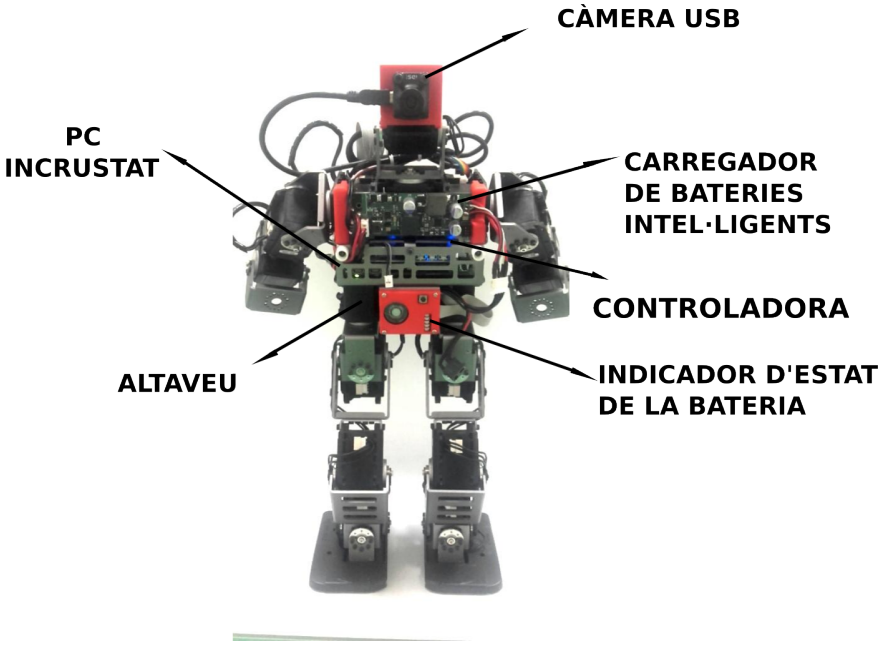


Figura 4.7: Vista frontal del robot DarwinOP

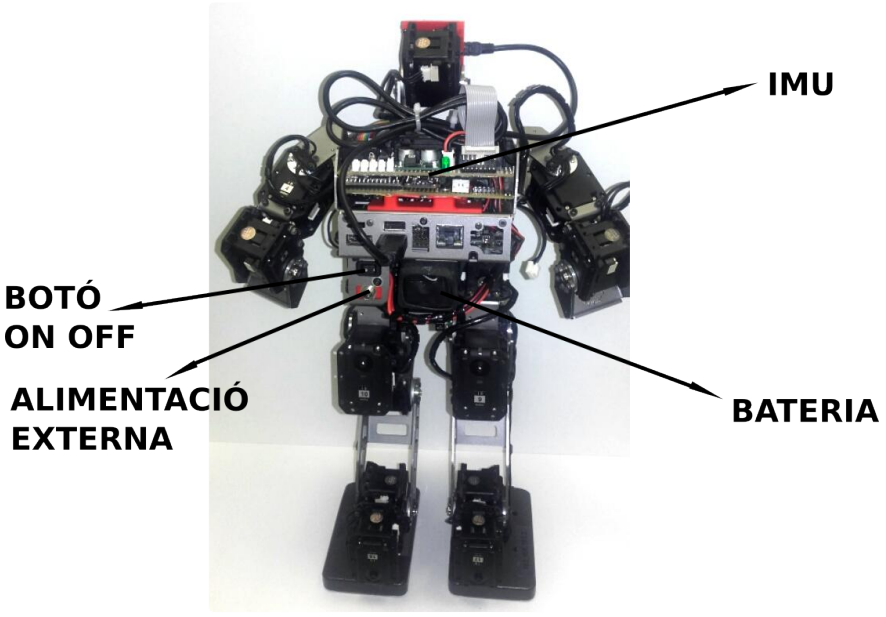


Figura 4.8: Vista posterior del robot DarwinOP

El robot es caracteritza per les següents especificacions:

CARACTERÍSTIQUES GENERALS:

- **PC incrustat/embeded:** Intel Atom Z530 1.6 GHz a 512MB RAM amb 4GB de flash SSD. Encarregat d'executar l'algoritme, processar les imatges de la càmera i enviar-li a la controladora el què ha de fer.
- **Controladora dirigida:** CM-730 ARM CortexM3 STM32F103RE 72MHz, encarregada de controlar el moviment dels servomotors [13] i gestionar els sensors i la resta de perifèrics excepte la càmera.
- **Sistema d'energia:** de 2200 mA, amb un carregador intel·ligent, i un adaptador de potència exterior.
- **comunicació CM730-servos**: es fa mitjançant una interfície sèrie que implementa el protocol dynamixel 1.0 [36].
- **comunicació PC - CM730**: es fa mitjançant una connexió USB que implementa el protocol dynamixel 2.0 [37].
- **20 mòduls actuadors:** En les cames: *6 graus de llibertat* , en els braços: *3 graus de llibertat* i al coll, *2 graus de llibertat.*

- **Mida:** conté unes mesures petites, 454,5 mm d'altura, 219,5 mm d'altura de les cames i, amb els braços oberts, mesura 271mm. Encara i ser petites, al ser un robot hominoide que realitzi algoritmes específics per interactuar amb l'entorn, les dimensions són apropiades per a la seva funcionalitat.
- **Mida del peu:** 104 mm de llargada i 66 mm d'amplada. Es tracta d'una superfície de 68,64 cm^2. Ha de ser adequada per poder suportar el pes del robot durant el seu moviment.
- **Pes:** El robot pesa 2,9 kg. És un bon pes per l'altura que té.
- **IMU's**: conté un giroscopi de 3 eixos i un acceleròmetre de 3 eixos.

Descripció dels servomotors:

Els servomotors que incorpora el DarwinOP es tracta del model MX-28[8] mostrat a la figura 4.9.

Figura 4.9: Servomotor MX-28 de dynamixel.

Aquests servomotors tenen les següents característiques:

- **Un algoritme de control:** PID
- **Pes:** 72g.
- **Dimensions:** 35.6mm x 50.6mm x 35.5mm.
- **Rang de moviment:** 0-360º. Encara que, limitats a 180º cap a la dreta i 180º cap a l'esquerra.
- **Voltatge (Alimentació):** 10-14.8V, amb un voltatge recomanat de 12V.
- **Sensor de posició:** Encoder absolut magnètic (12Bits, 360º), amb una resolució de 0.088º.
- **Tipus de protocol:** Utilitza el protocol dynamixel (1.0) sobre una línia de comunicacions en sèrie asíncrona [36].
- **Engranatges:** Metàl·lics.

4.2.2 Càmera USB

La càmera utilitzada és una càmera industrial USB 2 uEye XS, de la marca IDS [14] .

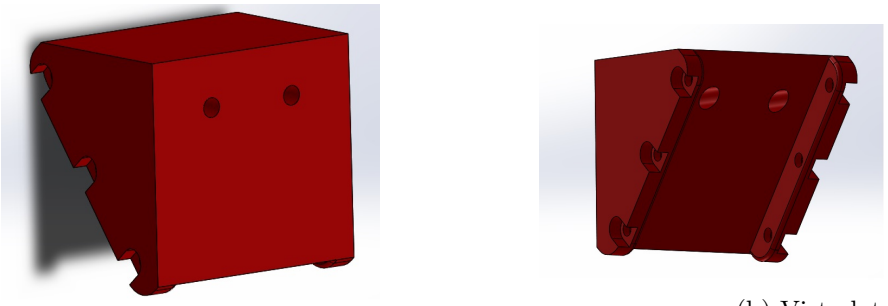
Figura 4.10: Càmera IDS

Les característiques més importants que conté la càmera són les següents:

- **Autofocus de 10 cm a** ∞
- **Connexió** USB 2.0
- **Framerate:** 15Fps : número d'imatges per segon màxim, es pot augmentar utilitzant binning a 5 MP.
- **Sensor**: CMOS ON de 5 megapíxels (2592 x 1944 píxels) de i una grandària de píxel de 1,4μmm,
- **Tamany**: 23 x 26,5 x 21,5 mm.

4.2.3 Suport càmera

S'incorpora en el robot una càmera nova, descrita en l'apartat anterior 4.2.2. Per això es dissenya un nou suport per situar-la al cap i poder-la orientar amb les articulacions de *pan* i *tilt* tal com es representa a la figura 4.12. El plànol de la càmera, es troba a l'annex **??**.



(a) Vista lateral dreta

(b) Vista lateral esquerra

Figura 4.11: Suport de la càmera IDS pel cap del robot.

A la figura 4.12 es mostra el disseny de la peça i com es fixa amb el robot.

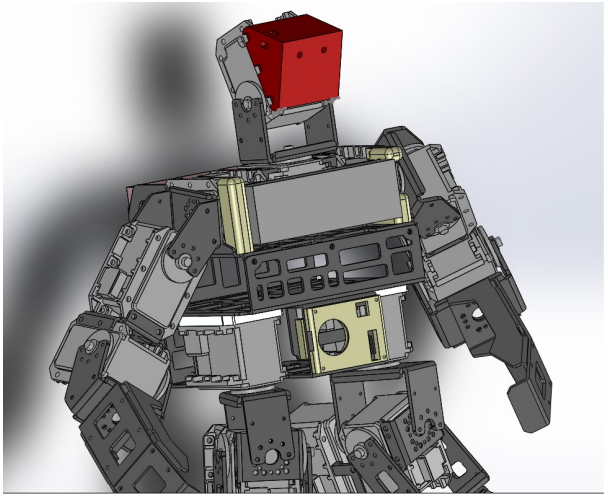


Figura 4.12: Distribució de la peça incorporada en el robot.

4.2.4 Driver/Firmware del robot darwinOP

Tot i que l'estructura mecànica del robot s'ha conservat majoritàriament, no ha passat el mateix amb el software. A la figura [4.6] es pot veure que hi ha una controladora de baix nivell (CM730)[5] que és la que té accés directe als servo Dynamixel MX28 [8], als sensors (giroscopi i acceleròmetre) i a d'altres perifèrics menys importants, la qual es comunica via USB amb un ordinador incrustat (FitPC2i)[10].

L'estructura de *software* original proporcionada per Robotis utilitzava la controladora de baix nivell principalment per fer de repetidor entre l'ordinador incrustat i els servos, re-enviant totes les comandes de moviment que generava el primer. A més a més també implementava les tasques de configuració i adquisició de dades dels sensors i d'altres perifèrics disponibles.

Amb aquesta configuració, era l'ordinador incrustat l'encarregat d'executar tant els algoritmes de control d'alt nivell, com les tasques de generació de les comandes de moviment per a cada servo per tal de dur a terme les tasques desitjades. La descripció detallada del software original de Robotis es pot veure a la pàgina següent [38]. A l'IRI, s'ha desenvolupat un nou firmware per a la controladora CM730 que implementa, no només les tasques de configuració i gestió dels diferents sensors i perifèrics disponibles, si no també les tasques de generació de les comandes de moviment per a cada servo, reduint així la càrrega computacional de l'ordinador incrustat.

A nivell de moviment, el nou firmware està construït al voltant d'un mòdul gestor (motion_manager) que té com a objectiu principal la gestió de diversos mòduls de moviment que executin diferents funcions. A part dels mòduls que ja proporcionava el *software* original d'execució de mo-

viment pre-determintas (accions), caminar i seguiment d'objectes amb el cap, s'ha afegit un mòdul per poder moure un conjunt arbitrari de servos a uns angles qualsevol amb unes certes velocitats i acceleracions.

El motion_manager també permet executar les següents accions:

- **Habilitar i deshabilitar** qualsevol servo per tal de reduir-ne el consum quan no sigui necessari.
- **Assignar** qualsevol servo a un únic mòdul de moviments del que hi hagi disponibles per poder assegurar que només rebrà ordre de moviment d'una única font.
- **Habilitar i deshabilitar** el balanceig del robot, així com ajustar els guanys del l'algoritme utilitzat. El balanceig fa servir la informació del giroscopi per poder modificar lleugerament la posició d'alguns dels servos de les cames per tal de fer que el moviment actual sigui més estable, i evitar així que el robot caigui.
- Com que existeixen petites diferències mecàniques entre dos servos qualssevol, també permet definir uns *offsets* per a cada servo per tal que la seva posició de zero coincideixi amb la posició desitjada.

A continuació es descriu amb detall les característiques principals de cada un dels mòduls de moviment existents:

- **Accions**: A la memòria no volàtil (EEPROM) de la controladora CM730 hi ha gravades un conjunt d'accions pre-determinades que pot executar el robot. Aquestes accions es poden identificar tant a través de la seva posició dins la memòria com també a través d'un nom, tot i que la interfície actual només permet la primera opció. Les funcions disponibles per aquest mòdul son:
  - Carregar l'identificador de l'acció desitjada.
  - Iniciar l'execució de l'acció.
  - Parar l'execució de l'acció.
  - Veure si l'acció s'està executant o no. Això és necessari per que el moviment no acaba immediatament després de parar-lo, si no que pot continuar durant una estona depenent del punt en que s'hagi parat.
- **Caminar**: El *software* original proporcionava un algoritme de caminar paramètric que permet generar de forma dinàmica el moviment de cada una de les cames a través d'un

conjunt de paràmetres de configuració i un conjunt de paràmetres de moviment (veure la secció 4.3.2 per a més detalls d'aquests paràmetres). El nou firmware implementa aquest mateix algoritme i permet realitzar les següents funcions:

* Configurar cada un dels paràmetres de moviment.
* Iniciar el moviment de caminar amb qualsevol conjunt de paràmetres de moviment longitudinal, lateral i de gir.
* Modificar de forma dinàmica qualsevol dels paràmetres de moviment
* Parar el moviment de caminar i veure quan el robot para de moure's de forma efectiva.

– **Seguiment del cap**: El *firmware* implementa un parell del controls PID per a les articulacions de *pan* i *tilt* que permeten configurar la manera com es segueixen els objectius dels angles de *pan* i *tilt*. La configuració de cada PID inclou les constants proporcional, integral i derivativa, així com un valor de saturació de la part integral de l'error. Les funcions que permet realitzar el nou *firmware* son les següent:

* Configuració dels paràmetres dels PID's de pan i tilt.
* Iniciar el seguiment de les consignes d'angle de pan i tilt proporcionades per l'usuari.
* Actualització en qualsevol moment de les consignes de pan i tilt.
* Parar el seguiment del cap. En aquest cas, quan es dona l'ordre de parar, el seguiment acaba de forma immediata. De totes maneres el firmware també permet veure si el seguiment està actiu o no.

– **Moviment d'articulacions**: Aquest nou mòdul permet definir fins a 4 grups independents de servos per poder realitzar moviments articulars. El moviment articular directe dels servos pot ser útil per a realitzar tasques de posicionament dels braços en l'espai per tal d'interactuar amb l'entorn utilitzant la seva cinemàtica inversa, i també per posicionar algun servo, o conjunt de servos, a una posició desitjada. Els grups es defineixen de forma dinàmica al assignar els moviments desitjats, i la única restricció és que un servo no pot pertànyer a més d'un grup.

Per a cada grup es poden realitzar les següents funcions:

* Càrrega de les posicions, velocitats i acceleracions per al conjunt de servos que es vulgui moure.
* Iniciar el moviment d'un grup.

* Parar el moviment d'un grup.
* Guiar i veure si algun servo d'un cert grup encara s'està movent o no. Com en el cas del seguiment del cap, aquesta funció no és imprescindible pel fet de que el moviment acaba immediatament després de donar l'ordre de parar.

A part de les funcions principals de moviment, el nou *firmware* també permet l'accés als perifèrics de la controladora a l'ordinador incrustat. En concret, de moment s'ha implementat suport per a tres mòduls:

- **GPIO's**: s'ofereix un conjunt de funcions que permeten controlar l'estat de cada un dels LED's de que disposa el robot, oferint funcions per encendre'ls i apagar-los, canviar l'estat actual i també fer-los parpellejar a una freqüència desitjada. Pels LED's RGB dels ulls del robot, també es permet poder escollir i canviarà el seu color. Pel que fa als polsadors, és possible veure el seu estat externament des de l'ordinador extern o bé utilitzar-los internament assignant funcions de *callback* per tal de realitzar les accions desitjades quan es pressionin.
- **Conversors AD**: el nou *firmware* permet també accedir a les lectures dels ports analògics de la controladora CM730 on s'hi poden connectar una gran varietat de sensors. A més a més també es pot canviar la temperatura de la controladora. La freqüència a la qual es fa l'adquisició de dades es pot configurar externament.
- **IMU**: tot i que les mesures del giroscopi i de l'acceleròmetre s'utilitzen internament a la controladora per poder realitzar l'algoritme de balanceig i poder detectar la caiguda del robot, la seva informació també es pot guiar des d'un ordinador extern. A més a més també és possible iniciar i guia el procés de cal·libració del giroscopi per tal d'eliminar els *offsets* en qualsevol moment.

Totes les funcionalitats de la controladora CM730 que s'han introduït fins al moment es fan accessibles a un ordinador extern a través d'un conjunt de registres que es poden llegir i escriure seguint el protocol Dynamixel 2.0 [37]. Per tant el *driver* de Linux del robot DarwinOP es simplifica enormement i només s'ha d'encarregar de donar les ordres desitjades escrivint als registres corresponents, i obtenir l'estat actual del robot i la informació dels sensors llegit dels registres corresponents.

Com es veurà en la següent secció 4.3, les interfícies ROS que hi ha disponibles pel *driver* de ROS del robot Darwin OP, permeten accedir de manera còmode a totes les funcionalitats que s'han introduït en aquesta secció.

4.3 Nodes i interconnexions:

En la figura 4.13 es mostra l'estructura de nodes utilitzats en simulació i la seva interconnexió a través de tòpics, serveis i accions.

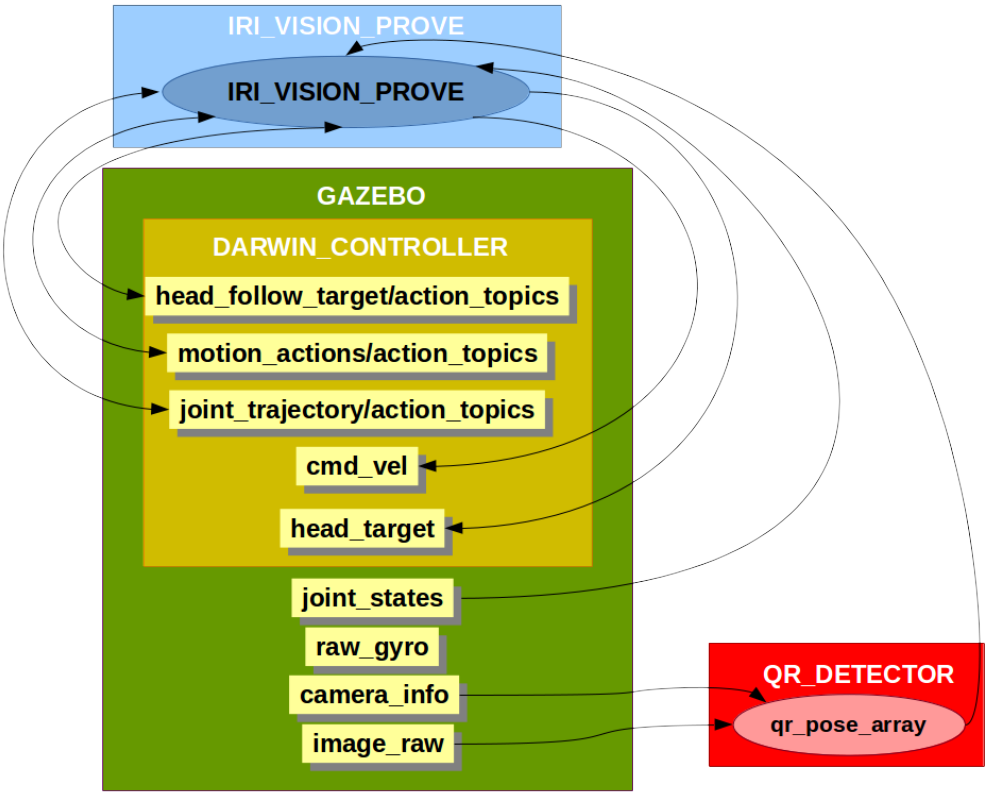


Figura 4.13: Distribució de nodes i interfícies de comunicació de ROS pel robot simulat.

Seguidament, a la figura 4.14 es representa la distribució de les mateixes interconnexions però ara en el robot real.

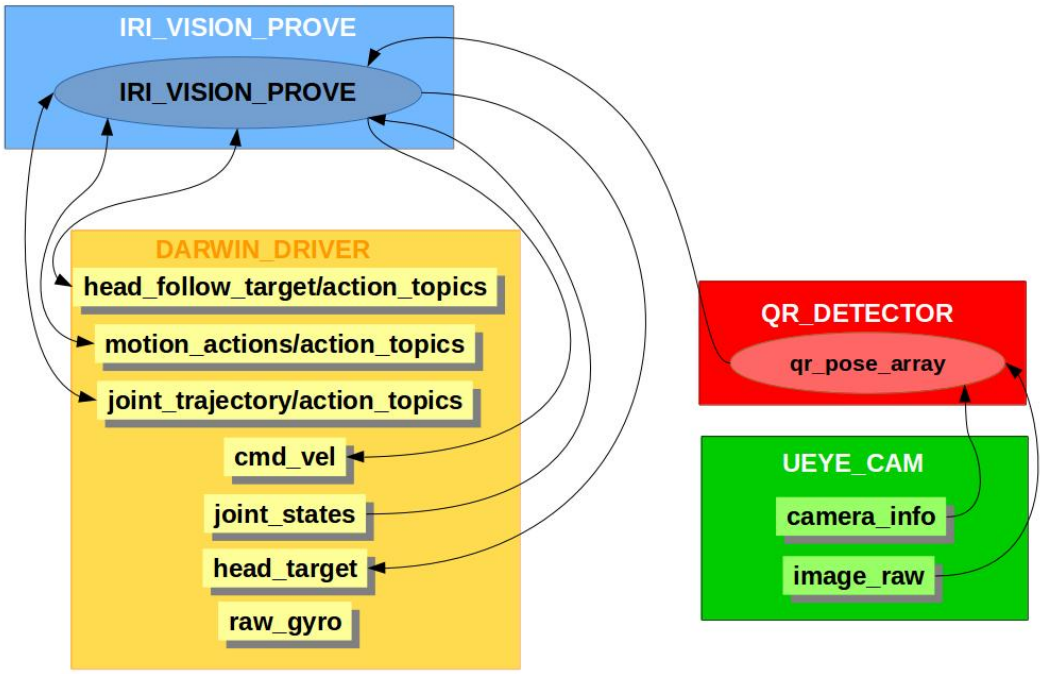


Figura 4.14: Distribució de nodes i interfçicies de comunicació de ROS pel robot real.

El node *iri_vision_prove* serà el node creat per a realitzar la prova de visió , s'explicarà al capítol 5. El node *darwin_controller*, conjuntament amb el simulador *gazebo* [29], per l'entorn simulat, o bé el node *darwin_driver* pel robot real, proporcionen tota la interfície per controlar el robot i per obtenir el seu estat. El node *QR_DETECTOR*, és l'encarregat de trobar codis QR en les imatges. Cal diferenciar la part de la càmera en el robot simulat i en el robot real. En el robot **simulat**, és el mateix *Gazebo* qui genera aquestes imatges, mentre que en el cas del robot real, figura 4.14, és el node *ueye_cam*, qui genera les imatges que processa el node qr_detector 4.3.6.

Els tòpics, serveis i accions, es descriuen en les següents seccions, classificats segons la seva funcionalitat que s'utilitzen en el la prova de visió són els següents:

4.3.1 Genèrics:

- **SET_SERVO_MODULES**

  El **servei set_servo_modules** és un servei encarregat de fixar un conjunt de servomotors del robot a un dels 4 mòduls de moviments diferents: al mode *joints*, al mode *head*, al mode

walk i, al mode *action*, en funció de l'ús que s'hi vulgui fer. Això permet que diferents servos treballin en diferents mòduls simultàniament.

La informació que es troba en el missatge, es troba definida en el missatge següent *humanoid_common_msgs/set_walk_params*. Aquest servei conté la següent informació:

```
string[] names
string[] modules
---
bool ret
```

El *names* farà referència al conjunt de servomotors, que es vulgui posar a un mode específic i, el *modules*, farà referència a quin mode es requereixi. El booleà *ret* retornarà *true* o *false* en funció si s'han fixat bé els paràmetres o no.

- **JOINT_STATES:**

  El tòpic *joint_states* és tracta d'un tòpic que dóna la informació de la posició, velocitat i, l'esforç dels servomotors en cada actualització. S'utilitza per saber l'estat dels servomotors i moure alguna articulació, com per exemple el cap, a partir de la posició en que estava .

  La informació que es troba en el missatge, està definida en *sensor_msgs/JointState*[25]

4.3.2 Per caminar:

- **CMD_VEL :**

  El tòpic **CMD_VEL** és l'encarregat de que el robot camini i giri cap a la direcció i sentit desitjat. Tal com s'observa en les imatges 4.13 i 4.14, el node DarwinOP hi està subscrit i, el node creat el publica. .

  La informació que es troba en el missatge, està definida en *geometry_msgs/Twist*[12]

- **SET_WALK_PARAMS**

  Per a que el robot camini bé, es crida al servei *set_walk_params*, què conté els paràmetres necessaris. La informació sobre cada paràmetre es pot trobar a la següent pàgina web [39]. A continuació es llisten els valors específics utilitzats pel robot:

```
Y.SWAP.AMPLITUDE = 0.02;
```

```
Z.SWAP.AMPLITUDE = 0.005;
ARM.SWING.GAIN = 1.5;
PELVIS.OFFSET = 0.05;
HIP.PITCH.OFFSET = 0.23;
X.OFFSET = -0.01;
Y.OFFSET = 0.005;
Z.OFFSET = 0.02;
A.OFFSET = 0.0;
P.OFFSET = 0.0;
R.OFFSET = 0.0;
PERIOD.TIME = 0.6;
DSP.RATIO = 0.1;
STEP.FB.RATIO = 0.28;
FOOT.HEIGHT = 0.04;
MAX.VEL = 0.01;
MAX.ROT.VEL = 0.01;
```

4.3.3 Execució d'accions pròpies del robot:

- **MOTION_ACTION:**

  L'acció motion_action s'utiliza per a que el robot pugui efectuar un moviment predeterminat, com podria ser una afirmació, negació, salutació,etc.

  La informació que es troba en el missatge, està definida a: *humanoid_common_msgs/humanoid_motionAction* i es mostra a continuació.

```
#goal definition
int32    motion_id
---
#result definition
bool     successful
---
#feedback
```

```
int32    current_page
int32    current_step
```

La motion_id és la acció predefinida que executarà, la qual ve donada per un valor enter.

El booleà successful retornarà *true* i s'ha completat l'acció i *false* si encara s'està executant. Els enters current_page i current_step donen la informació de la pàgina actual i el pas actual.

4.3.4 Seguiment del cap:

- **ACCIÓ HEAD_FOLLOW_TARGET :**

L'acció **head_follow_target**, permet iniciar o parar la funcionalitat de seguiment i un objectiu amb el cap.

La informació que es troba en el missatge està definida a: *humanoid_common_msgs/humanoid_follow_tar*
i es mostra a continuació:

```
#goal definition
float32 target_pan
float32 target_tilt
float32[] pan_range
float32[] tilt_range
---
#result definition
bool     successful
---
#feedback
float32 current_pan
float32 current_tilt
```

El target_pan i el target_tilt són els valors objectius de pan i tilt respectivament que s'ha de moure el cap i, el pan_range i el _tilt range són els rangs d'angles de pan i tilt acceptables per al moviment del cap.

- **HEAD_TARGET:**

El tòpic head_target s'utiliza a actualitzant l'objectiu de seguiment del cap, específicament com els angles de pan i tilt.

La informació que es troba en el missatge, està definida: *trajectory_msgs/JointTrajectoryPoint*[26]

- **SET PID :**

  Es tracta d'un servei que fixa els paràmetres de PID per a efectuar un moviment ràpid i precís.

  La informació del missatge es troba dins del fitxer *set_pid.srv*, i és la següent:

```
float32 p
float32 i
float32 d
float32 i_clamp
---
bool ret
```

El PID és un controlador que permet controlar el moviment dels servomotors. La constant p és la que fa que aquestos es moguin proporcionalment a la distancia final o *error*, la constant i, aporta exactitud i, la constant d, aporta rapidesa però, pot inestabilitzar el moviment. Aquest només s'ha implementat per realitzar el seguiment del codi QR . *HEAD_FOLLOW_TARGET* 4.3.4

4.3.5 Moviment articular:

- **ACCIÓ JOINT_TRAJECTORY:**

  L'acció Joint_Trajectory [15] s'utilitza per girar el cap tants graus com es vulgui amb una velocitat i un acceleració determinada.

  Tipus del missatge: *control_msgs/JointTrajectoryActionGoal*, i la seva informació es troba a [34].

4.3.6 Detector de codis QR:

- **QR_POSE:**

  El tòpic qr_pose proporciona informació sobre tots els codis QR que s'han trobat en cada imatge.

  Aquest missatge conté un vector, amb la informació dels diferents tags. Amb el missatge *tag_pose_array.msg*, amb la següent informació:

```
Header header
tag_pose[] tags
```

Dins de cada posició del vector, hi haurà la informació de cada codi QR que s'està veient, distribuïda de la següent forma:

```
Header header
string tag_id
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

4.4 Màquines d'estats

Per implementar l'algoritme de la prova de visió s'han utilitzat màquines d'estats [40].

Les màquines d'estats són estructures en forma de graf, les quals tenen estats i transicions. Es començarà amb un estat inici, els estats són segments de codi que realitzen alguna funció i, quan s'acaba un estat, es passara a un altre mitjançant una línia o transició cap a un altre estat, fins que s'arriba a l'estat final.

Aqueses màquines d'estats es crearan amb una jerarquia. Hi haurà la màquina d'estat principal, encarregada de crear l'algoritme necessari per efectuar la prova de visió amb èxit, anomenada **primària**. I, les màquines d'estats secundàries, agrupades en funcions les quals faran una funció concreta, tal com s'explicara en l'apartat 5.2.

En el cas de les funcions **secundaries**, sempre i quan siguin necessàries, es cridaran en forma de funció, retornaran *True* o *False* en funció si s'ha complert la màquina d'estats o no. Si apareix un **error exterior o la màquina d'estat s'ha complert amb èxit**, retornarà **True**. Per altra banda, es retornara **False** quan encara s'estigui **executant** la màquina d'estats. D'aquesta forma, la màquina d'estats primària, pot saber si s'ha acabat d'executar la funció o no, i passar al següent estat.

En l'explicació de cadascuna de les màquines d'estats, s'utilitzen diagrames d'estats organitzats de la següent forma:

Cada màquina d'estat tindrà un color fixat, en el cas de la màquina d'estat **secundaria**, els estats de la mateixa funció tindran el mateix color. En canvi, els estats de la màquina d'estat

primària que utilitzin les funcions de les màquines d'estats secundaries, utilitzaran els colors definits en les funcions secundaries juntament amb el propi. Si no n'utilitzen cap, mantindran el color específic de l'estat primari. Les transicions entre estats o *camins* seran dissenyats amb línies i fletxes descrivint la direcció i el sentit cap a on vagin. En cada transició s'indicarà la condició que s'ha de complir per fer-la. A la part inferior esquerra hi apareixerà una *llegenda*, tal com s'observa a la fig 4.15.

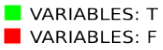


Figura 4.15: Llegenda de les màquines d'estats

El verd serà la condició correcta que permetrà passar al següent estat: **T: True** i, el vermell formarà part d'aquells camins que donin error en l'execució de l'estat, **F: False**.

4.5 Localització dels codis QR

Un codi QR[6] (*Quick Response*), codi de resposta ràpida , el qual permet guardar informació. Hi ha tres quadrats en les cantonades que permeten detectar la posició del codi.

Els codis QR que utilitzarem són codis QR quadrats, amb un costat de 0.084m el qual s'observa a la imatge 4.16 Els quals podran tenir les següents identificacions:

- *Turn45L* i *Turn45R*.
- *Turn90L* i *Turn90R*.
- *Turn135L* i *Turn 135R*.
- *Turn 180L* i *Turn 180R*

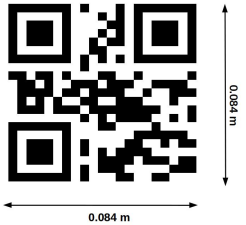


Figura 4.16: Codi QR utilitzat en la prova.

Es calculen les posicions *x,y,z* del centre del codi QR, respecte del centre de la càmera i el quaternió, què representa la orientació de la base del codi QR respecte del centre de la càmera. Aquesta informació vindrà donada pel tòpic *qr_pose*, al qual es subscriurà el node *iri_vision_prove* per obtenir la informació en cada actualització.

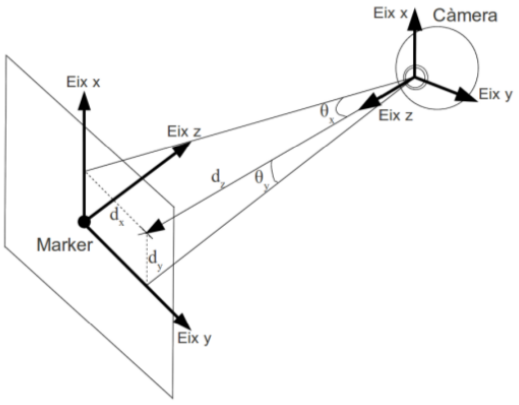


Figura 4.17: Posició relativa entre una càmera i un codi QR

La relació entre la càmera i el codi QR es farà segons la representació de la fig. 4.17. La pose ens retornarà la posició x ,y ,z del centre del codi QR respecte de la càmera. I, la orientació de la base del codi QR respecte la base de la càmera. I es calcula l'angle en què s'ha mogut el codi QR respecte de la càmera, amb els següents angles:

$$\theta_x = arctan2\frac{d_x}{d_z} \tag{4.1}$$

$$\theta_y = arctan2\frac{d_y}{d_z} \tag{4.2}$$

Aquests valors d'angles seran relatius, per això serà necessari sumar-li a les equacions 4.1 i 4.2 els valors de pan i de tilt actuals per obtenir la posició final o absoluta.

.

Part II

PART II

Capítol 5

PROVA DE VISIÓ

5.1 Descripció prova de visió

La prova de visió fou inspirada en el concurs CEABOT [4].

L'objectiu de la prova és detectar un codi QR, descodificar-lo i, girar tants graus en la direcció que digui la informació del codi QR. Cada codi QR indica la següent acció que ha de realitzar el robot. Hi ha 8 marcadors diferents que indiquen les rotacions de 45º, 90º, 135º i 180º a dreta o a esquerra, adherits sobre els obstacles tal i com mostra la figura 5.2

Hi haurà 8 obstacles, localitzats a intervals de 45º i a una distància variable del centre, tal com es distribueix en la fig. 5.1, els quals poden tenir codis QRs o no.

El robot iniciarà la prova al centre del camp, encarat al primer marcador. Haurà de descodificar-lo i interpretar-lo per poder arribar al següent.

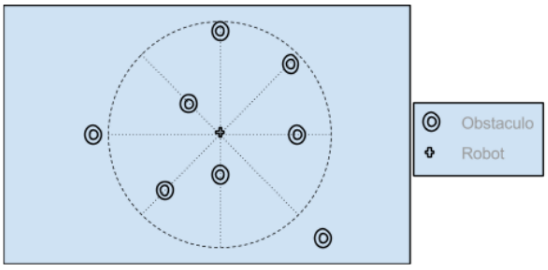


Figura 5.1: Exemple de distribució de codis QR per la prova de visió

El temps màxim disponible per aquesta prova es de 5 minuts. Començarà a contar quan el robot, després de realitzar una pausa de 5 segons, es posi en moviment i, realitzar la prova sense que l'usuari el pugui tocar.

Figura 5.2: Exemple d'obstacle amb un codi QR

5.2 Implementació prova de visió:

Per realitzar la prova de visió, es crea un *node* de *ROS* amb el nom de **iri_vision_prove**, que contingui les estructures de *software* necessàries, explicades en el capítol anterior. L'estructura del programa principal, s'implementa en la part del node *mainNodeThread* (ref. 4.1.3) en forma de màquina d'estat amb la jerarquia explicada en l'apartat 4.4, amb una freqüència de funcionament de 0,1 Hz.

Per tant, es crea una *màquina d'estats* **principal:** *States*, la qual s'encarregui d'inicialitzar la prova quan l'usuari ho demani, complir amb l'objectiu de cada estat utilitzant sempre i quan siguin necessàries, les màquines d'estats **secundàries** , la gestió dels possibles errors tant *externs* com *interns* que puguin aparèixer durant la seva execució i finalment, la fi amb èxit de la prova.

Els detalls de la implementació de la màquina d'estats **principal** s'explicaran a l'apartat 5.3 i els de les màquines d'estat secundàries, s'explicaran a l'apartat 5.4.

Dins del node, hi hauran dues *funcions de callbacks* , encarregades de processar la informació que es rep tant del robot com del detector de QR's

1. La funció **joint_states_callback**, subscrita al tòpic *joint_states*, explicat en l'apartat [4.3.1], donarà la informació de l'angle actualitzat de qualsevol servo. En particular s'utilitzarà per obtenir l'angle actual dels servos *j_pan i j_tilt* i així calcular *l'angle absolut* que hauran de girar aquestes articulacions.

2. Per altra banda, la funció **qr_pose_callback**, subscrita al tòpic *qr_pose*, explicat a l'apartat 4.3.6, procesa la informació dels QR's per obtenir la desviació horitzontal i vertical (en radians) de cada codi detectat amb el centre de la imatge, calculat en l'apartat 4.3.6, conté una llista amb la informació dels QR que està veient.

Tanmateix, existeix un vector, anomenat **exists**, el qual anirà guardant a la última posició, l'ultim codi QR descodificat, obtenint un vector ordenat per QR detectats i executats. D'aquesta forma, en cas que el robot perdi el QR podrà saber quin és l'últim codi detectat i el buscarà. Aquest vector també s'utilitza per poder filtrar QR's ja vistos quan se'n busca un de nou.

El codi de la prova es pot trobar a la següent pàgina: [33]

5.3 Màquina d'estat primària

En aquesta secció es parlarà de la implementació de la màquina d'estat primària amb la tipologia explicada en l'apartat 4.4. Aquesta màquina d'estat d'alt nivell es distribueix de la següent forma:

5.3.1 INIT

El primer estat, l'estat **INIT**, espera a que li arribi la notificació de què el botó esta pitjat, per començar l'execució de l'algorisme. Aquesta notificació es pot fer mitjançant la variable **config_.INIT** del *dynamic reconfigure* 5.5, o bé per un missatge de botons(simulació) o pitjant el botó fisic(robot real).

5.3.2 JT_FCTN

L'estat de **JT_FCTN** és l'encarregat de posicionar els servomotors *j_pan* i *j_tilt* a una bona posició perquè la càmera pugui detectar el primer codi QR. Aquest estat, utilitza la funció JT (*groga*), explicada en l'apartat [5.4.1]. Aquests paràmetres es troben i es poden modificar mitjançant les variables *pan_angle_init*, *tilt_angle_init*, *pan_velocity* i *tilt_velocity* 5.5 del *dynamic reconfigure*. .

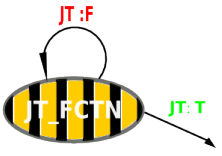


Figura 5.3: Estat JT_FCTN de la màquina d'estat principal

Per això, si la funció **JT** es realitza amb *èxit*, els servos arriben a la posició desitjada, es passa al estat següent. Si pel contrari, la funció JT encara esta *activa*, es quedarà en el mateix estat, fins que arribi al èxit.

La funció JT retorna *false* quan els servos encara s'estan movent, si retorna *true* serà que els servomotors s'han acabat de moure, o bé, hi ha hagut un error intern durant el moviment d'aquests,

per tant **no es possible gestionar possibles errors** tot i que en el cas que es produeixin es mostraran per pantalla.

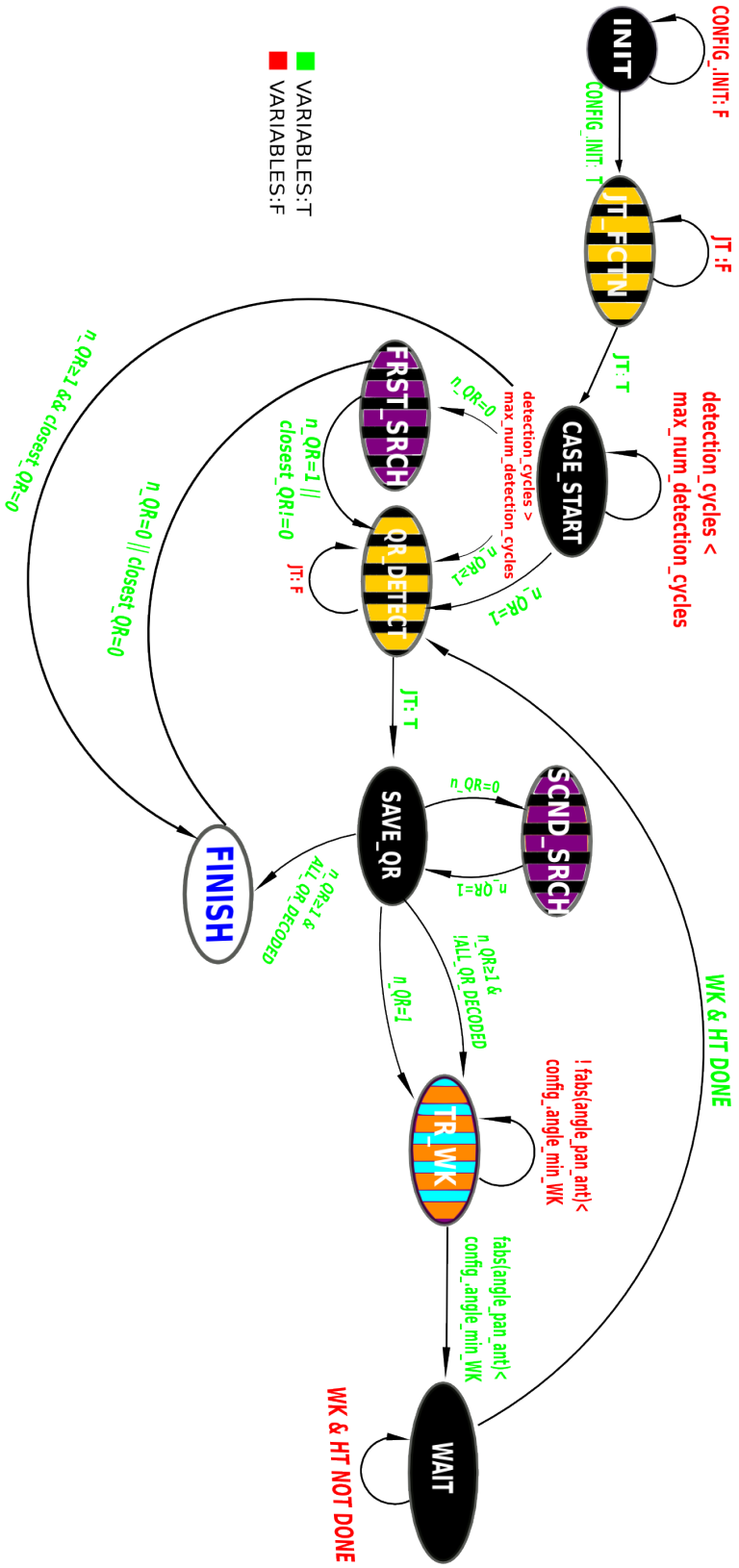


Figura 5.4: Màquina d'estat Primària

5.3.3 CASE_START

Després de moure el cap per centrar el *primer* codi QR, es descodifica el QR i es guarda al vector *exists*[5.2]. Per tant, es crearà l'estat **CASE_START** encarregat de guardar el primer codi QR descodificat.

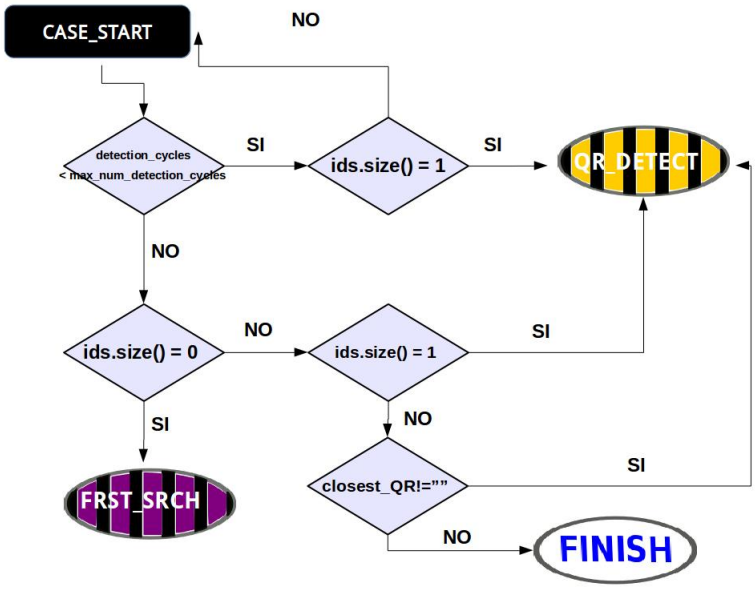


Figura 5.5: Diagrama de flux de l'estat CASE_START de la màquina d'estat principal

La variable **closest_QR**, la qual prové de la funció *get_closest_QR()* ,les variables **detection_cycles**, **exists** i la constant **ids.size()**.

La funció **get_closest_QR()** és una funció creada per retornar el codi QR de la llista de codis QR que s'estan veient amb l'angle entre el centre de la càmera i el centre del codi qr més petit, (com calcular-lo, s'explica en l'apartat 4.5). La variable **detection_cycles**, definida en el *dynamic reconfigure* [5.5] s'utilitza per saber quantes vegades s'entra dins de l'estat *CASE_START*. I, finalment, la constant **ids.size()**, prové de la funció *qr_pose_callback*, explicada en l'apartat 5.2, la qual informa del nombre de QR que la càmera detecta en cada *callback* o *actualització*. Les variables exists i ids.size(), venen definides en l'estat 5.4.1

Tal com es pot veure en el diagrama de fluxos de la figura 5.5, l'estat **CASE_START** inicia observant quantes vegades s'ha intentat fer la detecció fins al moment (nombre de deteccions <max_num_detection_cycles), en el cas que no s'hagi arribat al valor màxim i, només hagi detectat un codi QR, es passa al següent estat *QR_DETECT* guardant el codi qr detectat al vector *exists*,

si no es així, s'incrementa el nombre d'intents *detection_cycle++* i es torna a inicialitzar l'estat *CASE_START*.

Si es supera el nombre màxim d'iteracions i no es detecta cap codi QR, es passa al primer estat de cerca **FRST_SRCH**. Si no, es mira si s'ha detectat un codi QR (**ids.size()=1**). Si es detecta, es passa al següent estat **QR_DETECT** guardant el codi QR detectat al vector **exists**, si no es així, voldrà dir que es detecta 2 o més codis QR per tant, amb la funció **get_closest_qr** agafa el codi QR més pròxim a la càmera. Si es compleix, es passa al següent estat **QR_DETECT**, si no es així, s'acaba el programa anant a la funció final **FINISH**.

La càmera pot no detectar al primer instant una imatge o, bé capturar-la sense detectar cap codi QR; Per a solucionar-ho, la variable nombre de deteccions (*detection_cycle*) permet donar diversos intents a l'algoritme de detecció de QR's abans de procedir en cas que no es detecti cap codi.

5.3.4 FRST_SRCH

Aquest estat s'utilitza quan l'estat anterior *CASE_START* no troba ningun codi qr (*n_QR=0*) vist per la càmera.

Aquest estat utilitza la màquina d'estat secundària **search_QR**, explicada en l'apartat [5.4.5], la variable **exists**[5.2] i la constant **ids.size()**, explicades en l'estat anterior [5.3.3].

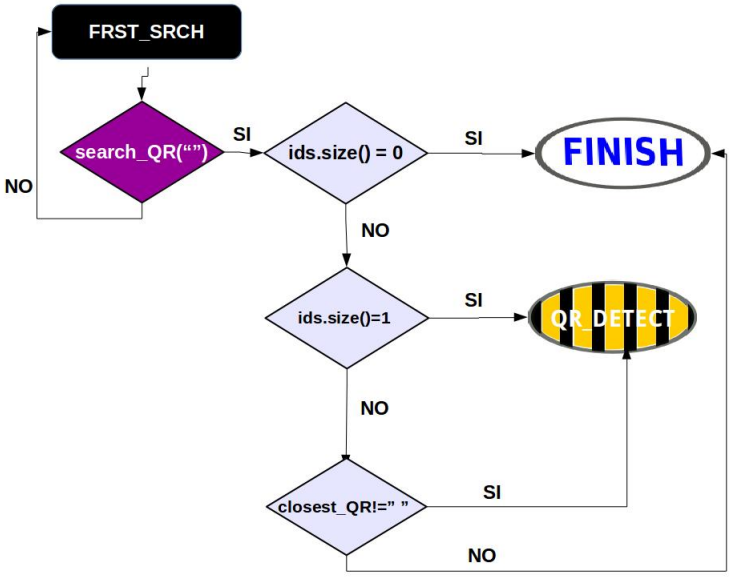


Figura 5.6: diagrama de flux de l'estat FRST_SRCH de la màquina d'estat principal

El funcionament és molt semblant al cas de l'estat anterior **CASE_START**, amb la diferència que s'executa la funció de cerca de forma paral·lela, la qual maximitza les possibilitats de trobar un codi QR i en cas de finalitzar sense trobar-ne cap, es passa a l'estat final **FINISH** 5.3.10.

Per això, si la funció **search_QR("")** retorna *false* es tornarà al mateix estat, ja que encara no ha trobat cap codi QR. Si retorna *true*, pot ser perquè hagi detectat un codi QR o bé, perquè ha realitzat tot l'algoritme de cerca sense èxit. Per això es mira si es detecta algun codi QR. Si no es detecta cap i per tant, (*ids.size()=0*), es passa a l'estat **FINISH** 5.3.10, si es detecta un codi QR (*ids.size()=1*), es passa a l'estat **QR_DETECT** i, finalment si es detecta 2 o més codis QR's es mira amb la funció *get_closest_QR* quin és dels codis que s'estan veient més pròxim al centre de la càmera. Si es així, es passa a l'estat **QR_DETECT** i, si no, es passa a l'estat **FINISH**.

5.3.5 QR_DETECT

L'estat **QR_DETECT** [fig. 5.7]és l'encarregat de girar el cap tants graus com el codi descodificat en l'estat anterior digui.

Utilitza la funció amb la màquina d'estat de baix nivell **JT**, explicada en l'apartat 5.4.1. L'angle de gir de *pan* s'obté de la funció **decode QR**, la qual descodifica el codi QR detectat i retorna l'angle de gir en radians(valors positius indiquen gir cap a l'esquerra). D'altra banda, l'angle de tilt es calcula segons l'equació 5.1, per tal de mantenir el màxim paral·lel possible el pla de la càmera al pla dels QR's.

L'equació de tilt que ens ho permet és la següent:

$$tilt = 0.707 + (tilt\_actual - 0.707) * cos(pan) \tag{5.1}$$

Les velocitats de *pan* i *tilt* es poden configurar a través de les variables *pan_velocity* i *tilt_velocity* del *dynamic reconfigure* 5.5. La màquina principal es quedarà en aquest estat fins que els servomotors del cap arribin a la posició desitjada.

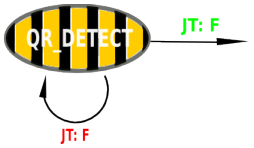


Figura 5.7: Estat QR_DETECT de la màquina d'estat principal

5.3.6 SAVE_QR

L'estat **SAVE_QR** és l'encarregat de trobar i guardar el codi QR que s'haurà de seguir en el següent estat *TR_WK* i executar-lo seguidament.

La principal diferència d'aquest estat, és troba en què abans d'agafar un codi QR, es mira que no s'hagi vist abans, i en cas afirmatiu es continua buscant. S'agafa el següent més proper dels que s'han detectat. Si tots els QR's que es troben ja s'han tractat, s'acaba la prova passant a l'estat FINISH

S'executa una variant del diagrama de flux de CASE_START fig. [5.5], utilitzant les mateixes variables amb els canvis següents:

- En el cas de **n_QR=1**, a més es mira si aquest codi està dins dels codis detectats anteriorment amb la funció *QR_exists*. Si es així, torna a l'estat actual, **SAVE_QR**, si no és així, va a l'estat *TR_WK*.

- *E*nlloc de passar a l'estat *QR_DETECT* passa a l'estat **TR_WK**.

- En el cas de cap QR (**n_QR=0**), s'executa un algoritme de cerca una mica diferent que *FRST_SRCH* [5.3.4], com es veurà a la secció [5.3.7] quan es descriu l'estat *SCND_SRCH*.

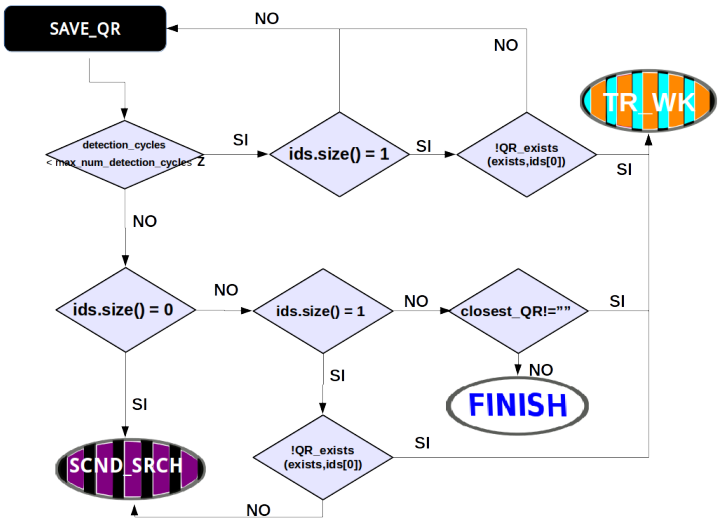


Figura 5.8: Diagrama de flux complementari de l'estat **SAVE_QR**

5.3.7 SCND_SRCH

L'estat de cerca segon, expressat en la figura 5.8 s'assembla molt al primer, vist en l'apartat 5.3.4. Les diferències són les següents:

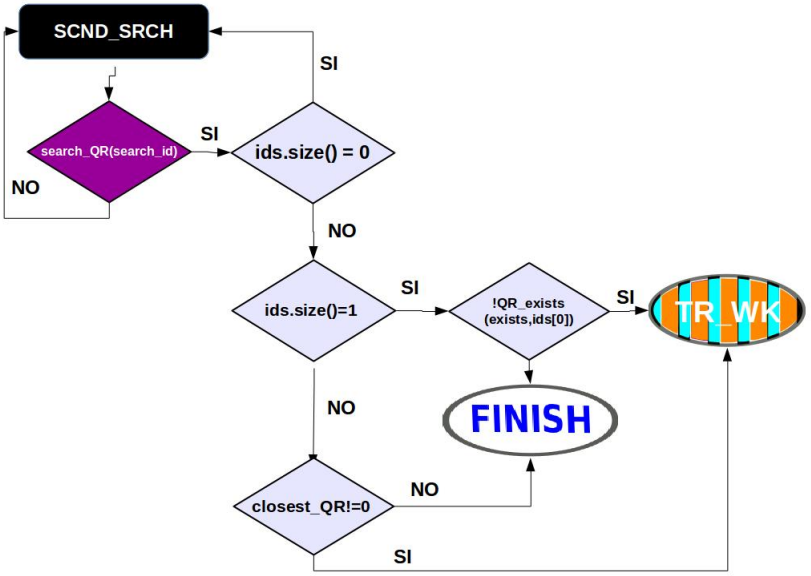


Figura 5.9: Diagrama de flux de l'estat SCND_SRCH

- Quan no veu cap codi QR **ids.size()=0**, busca el codi QR que s'ha perdut mitjançant la màquina d'estat secundària **search_QR**, explicada en l'apartat 5.4.5, passant-li com a paràmetre el codi QR que es busca. Finament, es tornarà al mateix estat *SCND_SRCH*.
- Si hi ha un codi QR **ids.size()=1**, mira si ja s'ha descodificat anteriorment amb la funció *QR_exists*. Si no es així, passa a l'estat **TR_WK**, si és així, es passa a l'estat **FINISH**, explicat en l'apartat 5.3.10.

.

5.3.8 TR_WK

L'estat **TR_WK** (*Traking_Walk*) té com a objectius executar un algorisme que segueixi amb el cap el codi QR detectat en l'estat anterior i alhora girar el cos el robot sobre el seu eix Z per aconseguir l'alineació del cos amb el cap per orientar el cap del robot al nou codi QR..

Aquest estat utilitza les màquines d'estat secundàries: **HT** i **WK**, explicades en els apartats 5.4.2 i 5.4.3 respectivament, la posició actual del codi QR desitjat *qr_pan_angle* i *qr_tilt_angle*,

l'índex del QR que estem tractant *qr_index*, obtinguts de la funció de callback *qr_pose_callback* explicada en l'apartat 5.2 i, la posició actual de les articulacions de pan i tilt obtingudes de la funció *joint_states_callback*, *angle_pan_ant* i*angle_tilt_ant*. Així com l'*angle_min_WK*, definit en el *dynamic reconfigure* 5.5, la qual serà una constant que definirà l'error mínim acceptable entre l'angle del cap i l'angle del cos, que per defecte, hauria de ser 0. I, finalment, *max_num_lost*, definida també en el *dynamic reconfigure*, serà el nombre màxim de vegades que pot estar aquest estat sense detectar cap codi QR.

El diagrama de flux encarregat de fer el seguiment del codi QR es representa la figura 5.10.

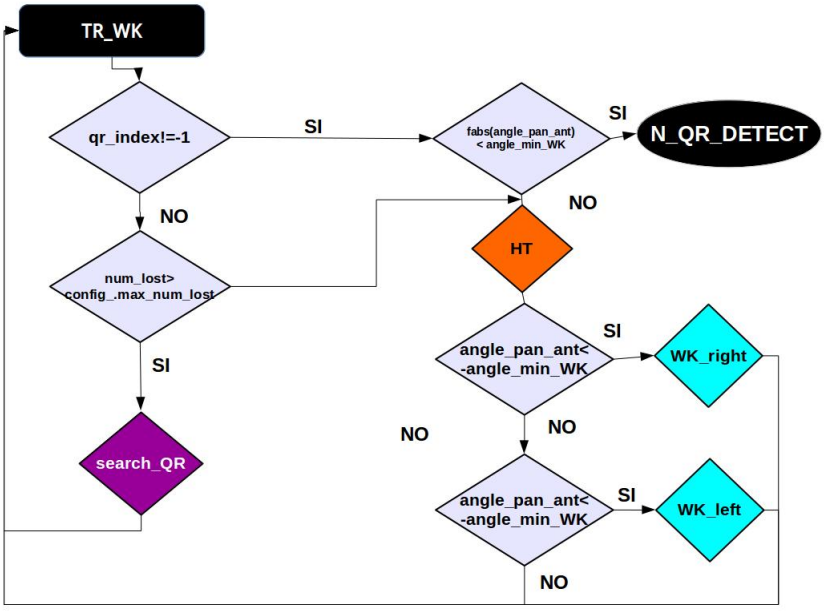


Figura 5.10: Diagrama de flux de l'estat **TR_WK** seguiment del codi QR.

El funcionament d'aquest estat és el següent: mentre l'angle actual de *pan*, el qual s'actualitza periòdicament), sigui més petit (en valor absolut) que un cert valor (angle_min_WK), s'executaran les funcions de caminar **WK** [5.4.3] i de seguiment del cap **HT** [5.4.2]. La funció HT farà que el cap sempre estigui mirant cap al codi QR desitjat, encara que el cos del robot es mogui. Quan l'error sigui més petit que l'angle_min_WK, es passarà al següent estat **N_QR_DETECT**

Paral·lelament, s'aplica la funció de caminar **WK** 5.4.3, per girar el cos en el sentit òptim per disminuir la variable *angle_pan_ant*. Es representa en el diagrama de flux de la fig. 5.10. Aquest s'encarregarà de minimitzar l'*angle_pan_ant* admitint un error mínim de *config_.angle_min_WK*.

Cal emfatitzar que el robot des de que s'envia l'ordre per parar de caminar fins que el robot para es necessita un *temps*, el qual pot provocar un l'error en l'inici del següent estat abans d'hora,

la solució es parlarà dins de l'apartat de la mateixa funció **WK**.

Pot passar que el robot tant en simulació com en la realitat perdi el codi QR, per això, tal com apareix en el diagrama de flux 5.10 si, es supera el nombre mínim d'intents per tornar a veure el codi QR perdut, *max_num_lost*, es passa a l'estat de cerca segon **SCND_SRCH**.

5.3.9 VERIFY_QR

Un cop assolit el nou QR, es passa altra vegada a l'estat QR_DETECT per executar l'acció indicada pel nou QR. En aquest estat s'espera que el robot acabi d'executar les accions de caminar i de seguiment del cap abans de continuar.

5.3.10 FINISH

Aquest estat vol dir que la prova de visió s'ha acabat, ja sigui per que no ha sigut capaç de descodificar els codis QR o bé, perquè ha realitzat tota la prova de visió satisfactòriament.

5.4 Màquines d'estat secundàries

Les funcions que contenen les màquines d'estat secundàries s'implementaran de la forma següent:

1. **JT** (float pos_pan,float pos_tilt,float vel_pan, float vel_tilt,bool start_JT, bool cancel_JT). La funció **JT**, explicada en l'apartat 5.4.1 s'utilitza per moure el cap sempre i quan sigui necessari arribar fins a un angle específic (l'angle es definirà a: *pos_pan, i pos_tilt*) amb una velocitat específica, (definida a: *vel_pan i vel_tilt*) en els paràmetres de la funció.

2. **HT** (float pos_pan,float pos_tilt,bool start_HT, bool cancel_HT). La funció **HT** , explicada en l'apartat 5.4.2 s'utilitza quan es requereix el seguiment d'un codi QR, donant-li com a paràmetre l'angle que ha de girar el cap per centrar-se amb el codi QR, (*pos_pan, pos_tilt*). Aquesta funció no acaba mai, s'ha de cancel·lar externament.

3. **WK**(float linear_x,float linear_y,float angular_z, bool start_WK,bool cancel_WK). La funció **WK**, explicada en l'apartat 5.4.3, s'utilitzarà quan es vulgui que el robot camini endavant, al costat i que giri, amb els paràmetres *linear_x, linear_y i angular_z* respectivament. Aquesta, per arribar al seu fi ha de ser cancel·lada externament.

4. **AC**(int type_action,bool startAC,bool cancelAC). La funció **AC**, explicada en l'apartat 5.4.4, s'utilitzarà quan es vulgui executar una acció pròpia del robot *darwinOP*, amb el nombre de

l'acció com a paràmetre de la funció *type_action*.

5. **search_QR**(std::string id_to_find). Finalment, la funció **Search_QR** s'utilitza per cercar un codi QR específic, si se li passa a la funció com a paràmetre *id_to_find* el id que es busca o bé, per cercar un QR qualsevol passant-li com a paràmetre de *id_to_find* buit (""").

Les màquines d'estat secundàries estan constituïdes pels següents estats:

5.4.1 JT - Joint Trajectory

La funció JT permet moure el cap (pan i tilt) fins a una consigna determinada amb una velocitat determinada.

La màquina d'estats secundària **JT** representada en la figura 5.11 consta dels següents estats:

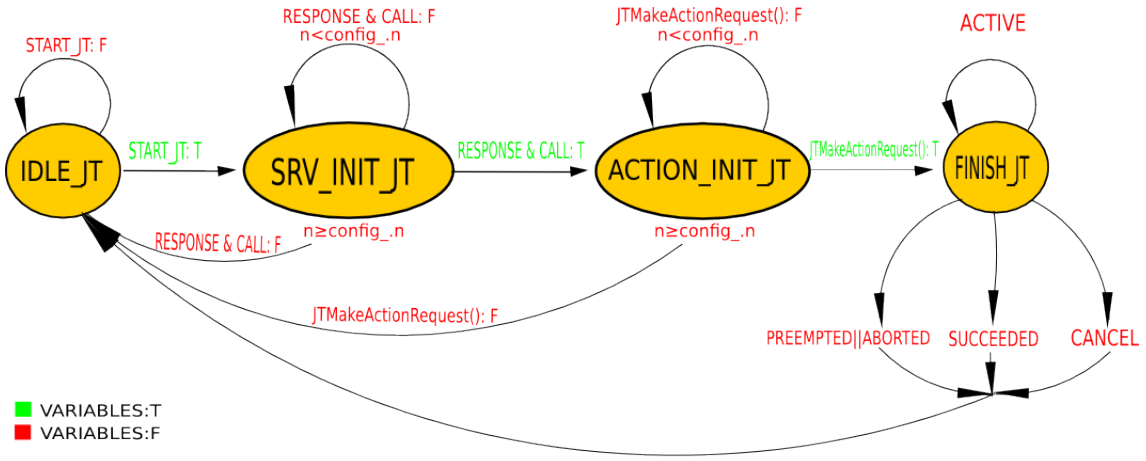


Figura 5.11: Màquina d'estat JT

IDLE_JT

És l'estat per defecte de la màquina d'estats, en si espera que el paràmetre d'entrada *start* sigui *true* per iniciar l'operació de la màquina d'estats secundaria. Quan el paràmetre *start* és *true*, la maquina d'estats passa al següent estat **SRV_INIT_JT**.

En aquest estat retorna *true* només si el paràmetre *START* és *false*. En els altres casos es retorna *false*.

SRV_INIT_JT

L'objectiu de l'estat **SRV_INIT_JT** es cridar el servei[4.1.2] *set_servo_modules* [4.3.1] dels servomotors *j_pan i j_tilt* en el mode *joints*, necessari per al moviment del cap.

La crida al servei pot fallar per dos motius, o bé perquè el servidor no rep la petició del cilent(mal mapping del servei) o bé, perquè la funció del servei no té èxit. En qualsevol cas només es passarà a l'estat següent quan la crida té èxit. En el cas en què s'intenti fer la crida al servei un cert nombre de vegades fins que sigui igual a *la variable num_set_servo_modules_calls del dynamic reconigure* sense èxit, la funció de la màquina d'estats secundaria torna a l'estat IDLE_JT i retorna *true*, en qualsevol altre cas retorna *false*.

ACTION_INIT_JT

Aquest estat és el que inicia el moviment del cap fins arribar a la posició i velocitat desitjades.

S'utilitza l'acció *joint_trajectory* amb els següents paràmetres d'entrada:

- **Joint_names**: es defineix el nom dels servomotors necessaris per moure el cap, *j_pan* i *j_tilt*.
- **Positions**: es defineix la posició objectiu, *pos_pan* i *pos_tilt*, els quals es defineixen en els paràmetres d'entrada de la funció.
- **Velocities**: la velocitat amb que es mouran els servomotors definits, *vel_pan* i *vel_tilt*.
- **Acceleration**: l'acceleració amb que es mouran els servomotors definits, *acc_pan* i *acc_tilt*.

Els paràmetres *pos_pan, pos_tilt, vel_pan i, vel_tilt*, es defineixen en els paràmetres de la funció **JT** i, els paràmetres de l'acceleració, *acc_pan* i *acc_tilt* es defineixen com a constants en el *dynamic reconfigure* 5.5.

La crida a la acció pot fallar per dos motius, perquè el servidor no rep la petició del client (mal mapping de la acció, o bé perquè la funció de l'acció no té èxit. En qualsevol cas només es passa a l'estat següent quan la crida té èxit. En el cas que s'intenti fer la crida a la acció un cert nombre de vegades, amb la variable *num_action_requests*, definida en el *dynamic reconfigure* 5.5 sense èxit, la funcó de la màquina d'estats secundaria torna a l'estat IDLE_JT i retorna *true*, en qualsevol altre cas retorna false.

WAIT_JT

L'estat **WAIT_JT** espera que l'acció finalitzi o es cancel·li externament.

Tal com s'ha explicat als coneixements previs, a l'apartat (4.1.2), hi ha diferents formes en què una acció pot acabar, per això en arribar a l'estat final **WAIT_JT**, en funció de com acabi l'acció es farà el següent:

- **ACTIVE**: Si l'estat de l'acció és *activa*, aquesta encara no ha acabat d'executar-se. Per tant, retorna *false* i l'estat següent es manté en l'estat actual **WAIT_JT**.
- **ABORTED**: En aquest cas l'acció finalitza a causa d'algun error intern de l'acció. En aquest cas la funció retorna *true* i es passa a l'estat IDLE_JT.
- **PREEMPTED |CANCEL** Es produeix quan l'usuari cancel·la voluntàriament l'acció.
- **SUCCEEDED**: En aquest cas l'acció acaba amb èxit i es retorna l'estat IDLE_JT retornant *true*.

Amb la implementació de màquines d'estat jeràrquiques, explicada en l'apartat 4.4, tant si l'estat acaba en error o, satisfactòriament es tracarà de la mateixa manera.

5.4.2 HT - Head Tracking

La funció HT, es cre per poder realitzar el moviment de tracking o seguiment del codi QR, movent el cap tant horitzontalment (*Pan*), com verticalment (*Tilt*), sense superar les restriccions del servomotor[+-180º].

La maquina d'estat **HT** consta de la següent distribució d'estats 5.12:

Els estats d'aquesta màquina d'estats segueixen l'estructura de la màquina d'estats secundaria anterior *JT* 5.4.1 amb les següents diferències:

IDLE_HT

Es comporta igual que l'estat *IDLE_JT* 5.4.1.

SRV_INIT_HT

L'objectiu de l'estat **SRV_INIT_HT** es cridar el servei *set_servo_modules* [4.3.1] dels servomotors *j_pan i j_tilt* en el mode **head (cap)**, necessari per al moviment del cap. Funciona igual que l'estat *SRV_INIT_JT* però en aquest cas les articulacions de *j_pan i j_tilt* s'associen al mòdul de head.

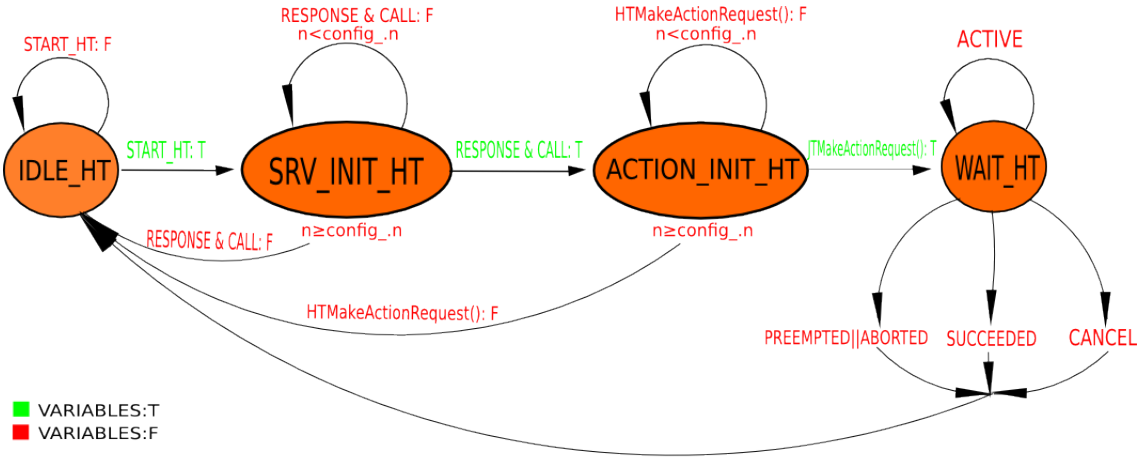


Figura 5.12: Màquina d'estat HT

ACTION_INIT_HT

L'estat **ACTION_INIT_HT** inicialitza una acció de seguiment del codi QR.

En aquest cas, s'utilitza l'acció *head_follow_target*, explicada en l'apartat 4.3.4 amb els següents paràmetres d'entrada:

- **Positions**: es defineix la posició objectiu, *pos_pan* i *pos_tilt* a través dels paràmetres d'entrada de la funció **HT** i, l'interval d'angle de *pan* i *tilt* en què els servomotors es mouen *pan_range* i *tilt_range* es defineixen en el *dynamic reconfigure* 5.5.
- **Range**: es defineix el rang de graus en què s'accepta el moviment dels servomotors en funció de les restriccions, tant en *pan* com en *tilt*, aquests rangs seràn de -PI a +PI.

WAIT_HT

Amb l'acció ja enviada, es passa a publicar al *tòpic* 4.1.2 **head_target** [4.3.4], actualitzant la informació dels angles de pan i tilt per poder seguir el codi QR.

Els paràmetres d'entrada d'aquest tòpic són els següents:

- **Positions**: s'utilitzen els mateixos paràmetres que en l'inici de l'acció *head_follow_target*.
- **Velocities**: la velocitat amb què es mouran els servomotors definits, *vel_pan_HT* i *vel_tilt_HT* seràn definides en el *dynamic reconfigure* 5.5.

Aquest estat s'executa de forma indefinida fins que es cancel·la externament utilitzant el paràmetre d'entrada cancel. Aquest paràmetre només es té en compte en aquest estat. Quan es cancel·li, es tornarà a l'estat IDLE_HT retornant *true*. En qualsevol altre cas retorna *false*

La informació s'ha d'enviar de forma continua perquè el *driver* del robot pari el moviment del cap si no rep cap missatge en 1 segon. Aquest estat s'executa de forma indefinida, fins que es cancel·la externament.

5.4.3 WK - Walking

La funció de WK controla el moviment del robot en totes direccions. Permet el moviment frontal, lateral, i el gir sobre si mateix.

La distribució del diagrama d'estats de l'estat WK és troba en la fig. 5.13.

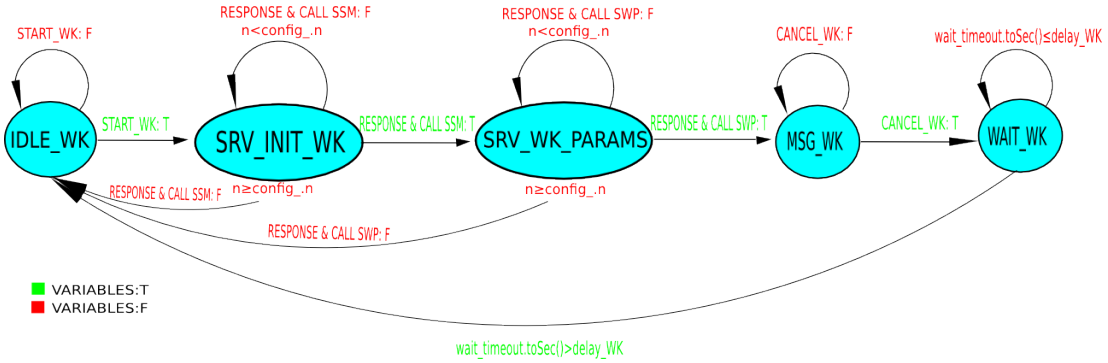


Figura 5.13: Màquina d'estat secundària WK

Els paràmetres que es defineixen en el missatge *cmd_vel_Twist_msg* venen dels paràmetres de la funció WK són els següents:

- Velocitat angular eix Z: **angular_z**; *en [rad/s]*
- Velocitat lineal X: **linear_x**; *en [rad/s]*
- Velocitat lineal Y: **en [rad/s]**; *linear_y*

IDLE_WK

Igual que en la secció 5.4.1.

SRV_INIT_WK

L'objectiu d'aquest estat és cridar el servei [4.1.2] *set_servo_ modules*[4.3.1] dels els servomotors de les cames *j_ankle_pitch_l/r",”j_ankle_roll_l/r",*
"j_shoulder_roll_l/r", "j_knee_l/r",”j_elbow_l/r",”j_hip_pitch_l/r",”j_hip_roll _l/r",

"j_hip_yaw_l/r","j_shoulder_pitch_l/r" en el mode **caminar (walking)**, necessari per al moviment del cos.

SRV_WK_PARAMS

L'objectiu de l'estat és cridar el servei[4.1.2] *set_walk_params* [4.3.2] fixant els paràmetres necessaris per a que el robot camini amb bones condicions. Es fixen amb els següents valors:

```
Y.SWAP.AMPLITUDE = 0.02;
Z.SWAP.AMPLITUDE = 0.005;
ARM.SWING.GAIN = 1.5;
PELVIS.OFFSET = 0.05;
HIP.PITCH.OFFSET = 0.23;
X.OFFSET = -0.01;
Y.OFFSET = 0.005;
Z.OFFSET = 0.02;
A.OFFSET = 0.0;
P.OFFSET = 0.0;
R.OFFSET = 0.0;
PERIOD.TIME = 0.6;
DSP.RATIO = 0.1;
STEP.FB.RATIO = 0.28;
FOOT.HEIGHT = 0.04;
MAX.VEL = 0.01;
MAX.ROT.VEL = 0.01;
```

Aquests paràmetres són definits a l'apartat coneixements previs 4.3.2. Són la causa de què el robot pugui caminar correctament, amb una bona compensació de la gravetat, fent que aquest no caigui. Aquests paràmetres venen de fàbrica i canviaran sempre i quan hi hagi una variació en el *hardware* del robot que afecti la distribució de pesos.

De la mateixa manera que en l'estat *set_servo_modules* 4.3.1 aquesta crida també pot fallar, per això es tornarà a intentar un cert nombre de vegades la crida, abans de tornar a l'estat IDLE_WK i retornar *true*. En cas de què els paràmetres de caminar es configurin correctament, es passa a l'estat MSG_WK.

MSG_WK

Aquest estat és l'encarregat de fer caminar al robot.

Mitjançant un missatge del tipus Twist [4.3.2], es publicarà al robot la informació de moviment desitjada.

La informació s'ha d'enviar de forma continua perquè el *driver* del robot pari el moviment si no rep cap missatge en 1 segon. Aquest estat s'executa de forma indefinida, fins que es cancel·li externament, igual que en l'estat de HT 5.4.2.

WAIT_WK

L'estat **WAIT_WK** és l'encarregat d'esperar que el robot acabi de moure's. El fet de que hi hagi un retard des de que s'ordena que el robot pari fins que efectivament pari, fa que s'hagi d'esperar un cert temps. Aquest dependrà de la velocitat, acceleració,etc.. abans de poder continuar el temps d'espera es pot ajustar a través de la variable **delay_WK**, del dynamic reconfigure 5.5. Només quan hagi passat el temps d'espera desitjat la màquina tornarà a l'estat IDLE i retornarà *true*.

5.4.4 AC - Action

La funció AC té com a objectiu realitzar diferents accions que existeixen dins del propi robot, com poden ser afirmar amb el cap, aixecar la mà saludant, etc. Poden ser de gran ajuda quan el robot acabi de descodificar el codi QR, o bé quan el robot inici o finalitzi la prova.

La distribució d'estats d'aquesta màquina d'estats secundaria secundària AC és la de la fig. 5.15:

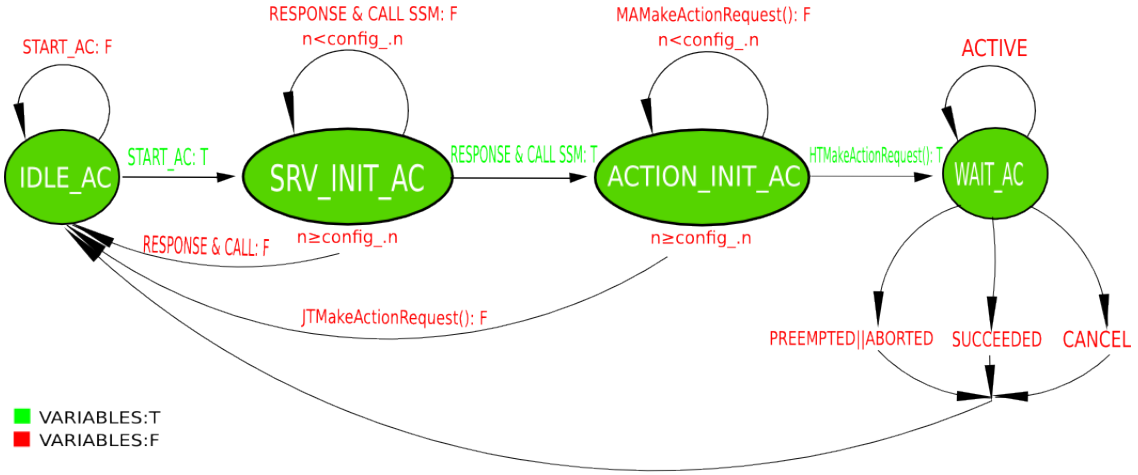


Figura 5.14: Màquina d'estat AC

Els estats d'aquesta màquina d'estats segueixen l'estructura de la màquina d'estats secundaria anterior *JT* 5.4.1 amb les següents diferències:

IDLE_AC

Igual que en la secció 5.4.1.

SRV_INIT_AC

L'objectiu de l'estat **SRV_INIT_HT** es cridar el servei[4.1.2] *set_servo_modules* [4.3.1] dels servomotors *j_ankle_pitch_l/r",j_ankle_roll_l/r", "j_shoulder_roll_l/r", "j_knee_l/r",j_elbow_l/r",j_hip_pitch_l/r",j_hip_roll _l/r", "j_hip_yaw_l/r",j_shoulder_pitch_l/r"* en el mode **acció (action)**, necessari per a realitzar alguna acció.

ACTION_INIT_AC

L'estat **ACTION_INIT_AC** inicialitza una acció de *moviment d'acció*.

A diferència que en les màquines d'estat anteriors **JT** i 5.4.1, s'utilitza l'acció *motion_action*, explicada en l'apartat 4.3.3 amb el paràmetre d'entrada**motion_id**, el qual defineix quina de les accions possibles es vol realitzar *type_action*, definida en el dynamic reconfigure 5.5. I, declarada com a paràmetre de la funció **AC**.

WAIT_AC

L'estat **WAIT_AC** fa la mateixa funció que l'estat **WAIT_JT**, espera que l'acció finalitzi o es cancel·li externament.

Farà el mateix que en la màquina d'estats anterior, **JT** 5.4.1, canviant el nom dels paràmetres als d'aquesta màquina d'estat. Per tant, en lloc de passar a *IDLE_JT*, es passarà a *IDLE_AC* i, si s'ha de quedar a l'estat, es quedarà a l'estat actual.

5.4.5 cerca_QR

La funció *cerca_QR* o bé busca algun codi QR específic o bé qualsevol, tant en casos on se sap el codi QR que es busca com en casos quan no es sap el que es busca. També, fa que la càmera es centri amb el codi QR trobat.

Es tracta d'una funció què conté una màquina d'estats secundària *cerca_QR* amb diferents estats tal com es representa en la fig. 5.15.

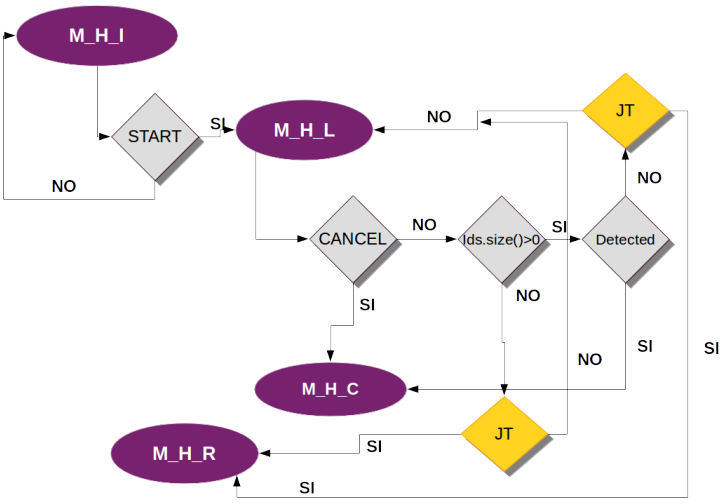


Figura 5.15: Màquina d'estat secundària cerca_QR

Aquesta conté les variables *START* i *CANCEL* les quals venen donades per la funció cerca_QR i el vector ids ja explicat. En aquest cas cal saber quin era l'angle del robot amb què es va entrar per calcular l'angle absolut que ha de girar la càmera. Finalment, la variable *search_num_steps*, farà referència al nombre màxim de iteracions de cerca que es vol fer i estarà definida al dynamic reconfigure 5.5.

Aquesta inicia en l'estat *MOVE_HEAD_IDLE* i, mira si esta activa o no (start=*true*). Si s'ha inicialitzat, comença la cerca per l'estat *MOVE_HEAD_LEFT*, si no es així, torna a l'estat inicial *MOVE_HEAD_IDLE*. A continuació es passa a l'estat *MOVE_HEAD_LEFT*, aquest mirarà primer si la cerca ha estat cancel·lada. Si no ha estat cancel·lada, mirarà si el nombre de codis QR que es detecten són més grans que 0, si es així, es passa a buscar el que està més centrat amb l'estat *MOVE_HEAD_CENTER*, si no es així, continuarà la cerca.

Després, quan va a l'estat *MOVE_HEAD_RIGHT*, fa exactament que en l'estat *MOVE_HEAD_LEFT* quan es detecti més d'un codi QR i no sigui el que estem buscant, però girant cap a la dreta ara. Si s'acaba el moviment, i no es supera el nombre de iteracions permeses *search_num_steps* es passarà altra vegada *MOVE_HEAD_LEFT*. Si es superen, es tornarà a l'estat inicial *MOVE_HEAD_IDLE*.

Si s'entra a l'estat *MOVE_HEAD_CENTER*, es centra la càmera del robot amb el centre del codi QR. Si aquest ja s'ha centrat, es passa a l'estat *MOVE_HEAD_STOP*, que para el moviment de centrar la càmera amb el codi QR i passa a l'estat inicial *IDLE*

5.5 Dynamic Reconfigure

En la següent taula s'expliquen les diferents variables que s'han utilitzat, amb el nom *NAME*, el tipus de la variable *TYPE*, el valor per defecte, *DEFAULT* i, el rang de valors acceptable *MIN* i *MAX*.

NAME	TYPE	DEFAULT	MIN	MAX
INIT	bool_t	False	-	-
max_num_set_servo_modules_calls	int_t	5	0	15
max_num_action_requests	int_t	5	0	15
walk_wait	double_t	0.9	0	10
pan_search_step	double_t	0.15	0	0.707
tilt_search_step	double_t	0.05	0	0.707
search_speed	double_t	0.5	0	3.14159
search_num_steps	int_t	5	1	10
max_num_lost	int_t	5	1	10
pan_angle_init	double_t	0.0	-1.5	1.5
tilt_angle_init	double_t	0.9	-1.5	1.5
pan_velocity	double_t	2.0	-3	3
tilt_velocity	double_t	2.0	-3	3
pan_acceleration	double_t	0.3	-3	3
tilt_acceleration	double_t	0.3	-3	3
vel_pan_HT	double_t	1.5	-3	3
vel_tilt_HT	double_t	1.5	-3	3
max_num_detection_cycles	int_t	10	0	15
angle_min_WK	double_t	0.2	-1.5	1.5
type_action	double_t	24	0	255

Taula 5.1: Taula de les variables del Dynamic Reconfigure utilitzades.

.

Capítol 6

Proves i Resultats

Utilitzant el simulador descrit en l'apartat 4.1.2, s'ha realitzat la prova del CEABOT[4] descrita en l'apartat 5.1. Per comprovar l'eficiència de l'algoritme, es necessari provar-ho tant amb el robot real com amb el robot simulat en tres camps diferents, canviant els codis QR's que s'utilitzen, augmentant la dificultat en cada prova.

L'objectiu d'aquest apartat és veure experimentalment l'efectivitat de l'algoritme tant en el robot simulat com en el robot real i comparar la implementació en els dos casos per treure les conclusions pertinents i els possibles canvis a millorar en projectes futurs.

Els vídeos de les proves es poden trobar a:

6.1 Prova amb dificultat simple:

En aquest cas es vol veure si l'algoritme és capaç de realitzar una prova senzilla, posant obstacles amb codis QRs de gir de 45 i 90º, evitant els QR de 135 i 180º. I, així detectar quins errors bàsics realitza l'algoritme del robot.

6.1.1 Descripció de la prova:

En la primera prova, el robot començarà la prova observant el codi QR *Turn45L* i, continuarà la successió següent com a la figura 6.1.

L'objectiu d'aquesta prova és realitzar els girs respectius segons els codis QR que es vagin trobant tal com es mostra en la figura 6.1, observar els diferents errors que surtin i treure les conclusions respectives.

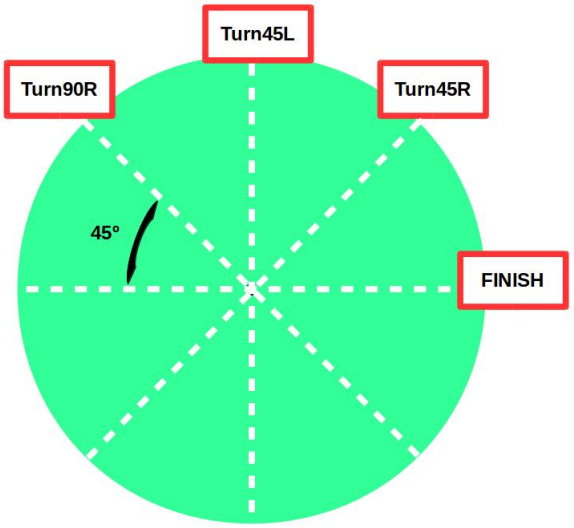


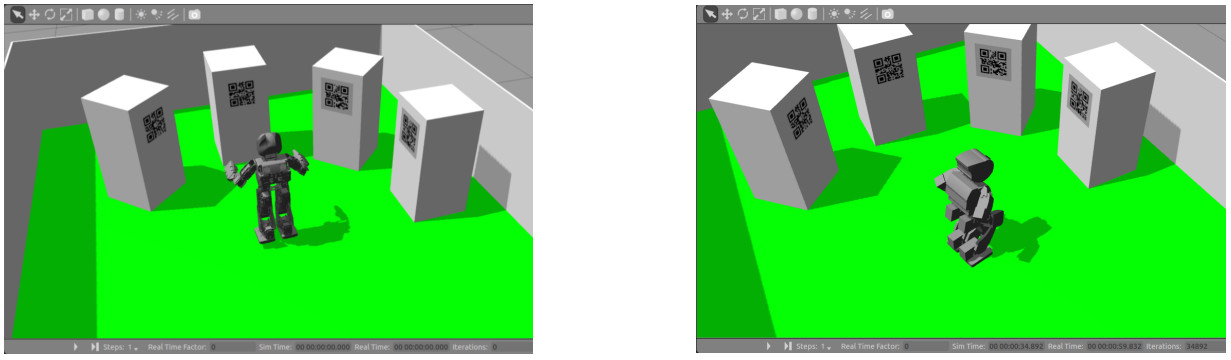
Figura 6.1: Distribució en l'espai dels codis QR's del primer experiment.

6.1.2 Resultats de la simulació:

La taula de resultats de simulació la primera prova és la següent:

Concepte	Prova1	Prova2	Prova3
Temps Simulació [s]	43.67	43	43
Nombre de vegades que perd el codi QR	0	0	0
Acaba la prova?	Si	Si	Si

Taula 6.1: Resultats de la primera prova amb simulació.



(a) Inici (b) Fi

Figura 6.2: Posició inicial i final en la realització de la primera prova del robot simulat.

En aquest cas, la prova de simulació s'executa amb 43 segons amb èxit, un temps molt bo per realitzar tota la prova. Aquest acaba amb èxit i no ha d'executar l'estat de cerca ninguna vegada,

ja que en ninguna descodificació es perd. Per tant, es pot assegurar que l'algoritme és correcte i funciona en perfectes condicions.

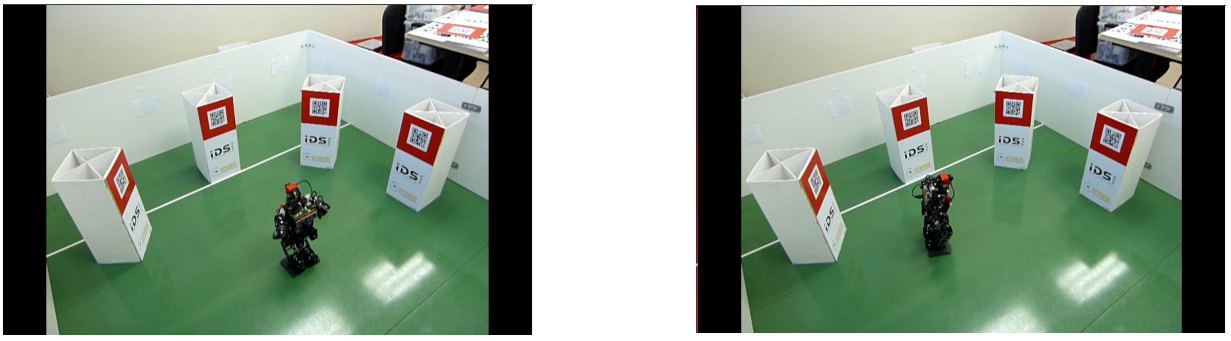
Pel que fa el moviment del robot, s'observa en les figures 6.2 com el punt en què el robot inicia i acaba, es diferència de poc espai. Per tant, el robot no gira sobre si mateix, sinó que fa una trajectòria el·líptica.

6.1.3 Resultats del robot real:

La taula de resultats de la primera prova és la següent:

Concepte	Prova1	Prova2	Prova3
Temps Simulació [s]	240	172.2	123
Nombre de vegades que perd el codi QR [45L]	0	0	0
Nombre de vegades que perd el codi QR [90R]	4	8	3
Nombre de vegades que perd el codi QR [45R]	5	6	4
Nombre de vegades que perd el codi QR [FINISH]	4		1
Acaba la prova?	Si	Si	No

Taula 6.2: Taula dels resultats de la primera proba amb el robot real



(a) Inici (b) Fi

Figura 6.3: Posició inicial i final en la realització de la primera prova del robot real.

En aquest cas, la prova s'executa amb un temps de 240, 172.2 , 123 *segons*, aquest temps és 5.63 , 4.04 i, 2.86 vegades més gran que el temps de simulació. Les raons són les següents:

Primerament, quan el robot camina perd el codi QR a causa del gran balanceig i la forma de caminar del robot què provoca que es desequilibri i a vegades camini cap al costat on no toca. El fet de perdre molt el codi QR i que el robot no giri sobre si mateix, provoca que després hagi de realitzar molts més algoritmes de busca.

Encara que el temps sigui molt més gran respecte al de simulació, l'algoritme s'executa amb èxit.

6.1.4 Com arreglar els errors:

El problema prové de la forma de caminar del robot. Ja que en la simulació també passa, vindrà d'alguna compensació de pesos (ja que la càmera pesa considerablement) o bé d'algun altre lloc, ja sigui que els paràmetres de caminar estiguin malament.

6.2 Proba amb dificultat mitjana:

En aquest cas es vol veure si l'algoritme és capaç de realitzar una prova amb una dificultat normal, posant obstacles amb codis QRs de gir de 45 i 90 i, 135 º evitant els QRs de 180º prioritzant la direcció de gir cap a la dreta. I, així detectar quins errors bàsics realitza l'algoritme del robot i, si la direcció de gir te efecte o no.

6.2.1 Descripció de la prova:

En la segona prova, el robot gira seguint la distribució de codis QR, començant observant el codi *Turn45R* girant aquest angle i detectant el següent I, així successivament, tal com s'explica a l'apartat **??**, tal com es mostra a la figura 6.4.

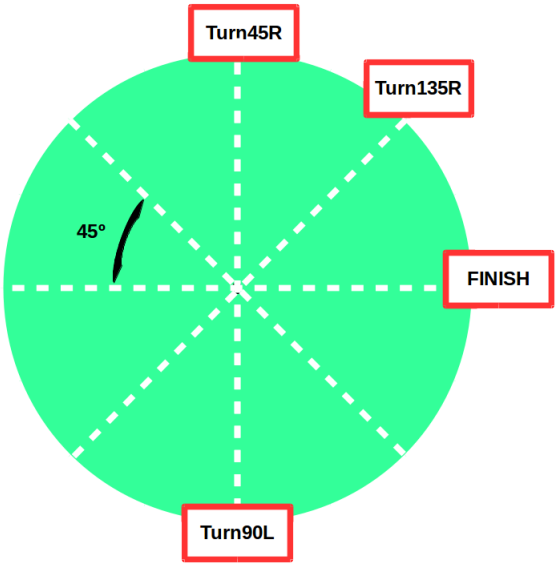


Figura 6.4: Distribució en l'espai dels codis QR's del segon experiment.

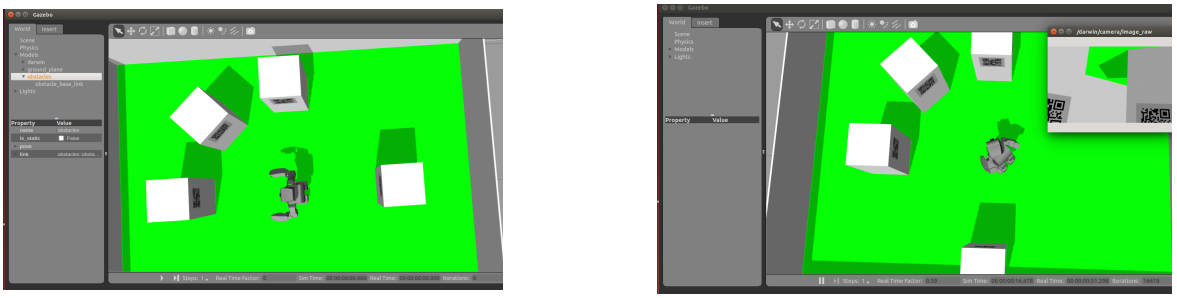
6.2.2 Resultats de la simulació:

La taula de resultats de la segona prova de simulació és la següent:

La taula de resultats de simulació de la segona prova és la següent:

Concepte	Prova1	Prova2	Prova3
Temps Simulació [s]	62.4	23.4	64.2
Nombre de vegades que perd el codi QR T90L	0	1	0
Acaba la prova?	Si	No	Si

Taula 6.3: Taula de resultats de simulació de la segona prova



(a) Inici (b) Fi

Figura 6.5: Posició inicial i final en la realització de la segona prova de simulació del robot real.

En aquest cas la prova de simulació s'executa amb un temps al voltant de 60 segons. El temps és més gran degut a què el robot ha de girar més, fins a 180º respecte la posició inicial. Aquest es perd en un cas, degut a que no detecta bé el codi *"Turn90L"* i, en entrar a la cerca, detecta primer el codi final *"FINISH"*, fet que acabi la prova sense èxit.

Pel que fa el moviment del robot, ja s'observa en les figures 6.5 com el punt en què el robot inicia i acaba no són els mateixos, per tant, es manté amb la trajectòria el·líptica que es tenia en el cas anterior.

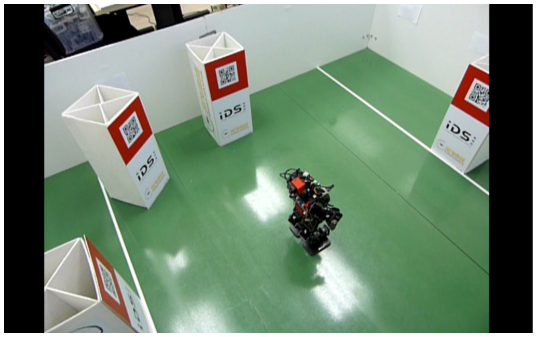
6.2.3 Resultats del robot real:

La taula de resultats de la segona prova amb el robot real és la següent:

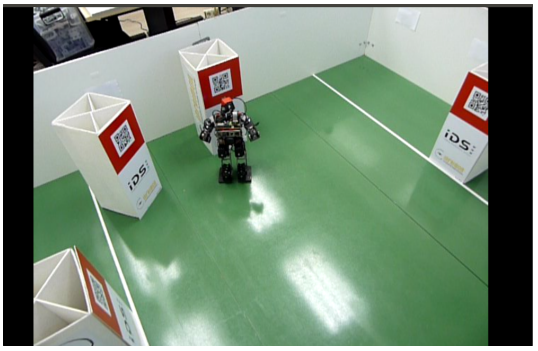
Concepte	Prova1	Prova2
Temps Simulació [s]	225.6	277.8
Nombre de vegades que perd el codi QR [45R]	-	-
Nombre de vegades que perd el codi QR [135R]	4	2
Nombre de vegades que perd el codi QR [90L]	26	Ajuda
Nombre de vegades que perd el codi QR [FINISH]	6	2
Acaba la prova?	Si	No

Taula 6.4: Taula de resultats de la segona prova amb el robot real.

En aquest cas, la prova s'executa amb un temps de 225.6 i 277.8 *s*, aquest temps és quatre vegades més gran que el temps de simulació. Les raons són les següents:



(a) Inici



(b) Fi

Figura 6.6: Posició inicial i final en la realització de la segona prova del robot real.

Es produeixen les mateixes raons que en la prova anterior 6.1, però, se n'afegeixen de noves:

Primerament, el cable de la càmera dificulta la visió en la descodificació i el el seguiment del codi QR *"Turn90L"*. Seguidament, la gran distància entre el codi *"Turn135R"* i el codi *"Turn90L"*, dificulta la descodificació del codi *"Turn90L"*.

En un dels casos, falla perquè en la cerca perd el codi QR. Això és produeix a causa del moviment el·líptic del robot, el qual fa que s'aproximi més als codis QR i, que la càmera no detecti el codi QR, ja que esta fora del seu abast.

6.2.4 Com arreglar els errors:

El cable que molesta es pot arreglar posant-li un altre sistema de fixació del cable.

El fet de què el robot estigui molt prop del codi QR, es podria arreglar posant a l'estat de cerca que tirés un pas enrere en cada iteració, i fer un petit estat de cerca en cada una . Així, seria més fàcil la seva detecció i es perdria menys temps en la cerca.

6.3 Prova amb dificultat elevada:

En aquest cas es vol veure si l'algoritme és capaç de realitzar una prova complexa posant obstacles amb codis QRs de gir de 45 i 90 i, 135 º evitant els QRs de 180º prioritzant la direcció de gir cap a la esquerra. I, així detectar quins errors bàsics realitza l'algoritme del robot.

6.3.1 Descripció de la prova:

En la tercera prova, el robot gira amb la distribució de codis QR com es mostra a la figura 6.7.

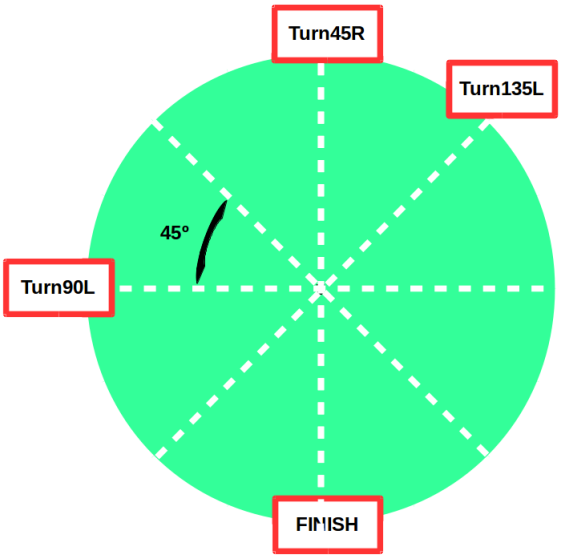


Figura 6.7: Distribució en l'espai dels codis QR's del tercer experiment.

6.3.2 Resultats de la simulació:

La taula de resultats de la simulació de la tercera prova és la següent:

Concepte	Prova1	Prova2	Prova3
Temps Simulació [segons]	190.8	241.2	196.2
Nombre de vegades que perd el codi QR T90L	3	3	2
Acaba la prova?	No	No	Si

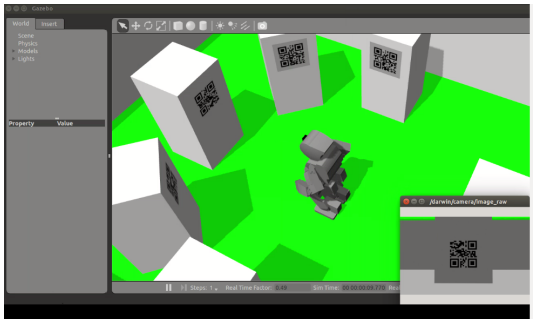
Taula 6.5: Resultats de la simulació de la tercera prova

En aquest cas, la prova s'executa amb un temps al voltant de 210 segons, es nota la magnitud del camp. El temps és mes gran degut a què el robot té molta dificultat per detectar el codi *"Turn90L"*. S'executa la cerca, però el robot arriba tan aprop del QR, que no el pot veure, per tant s'opta per ajuda-lo tant en la simulació com en la realitat .

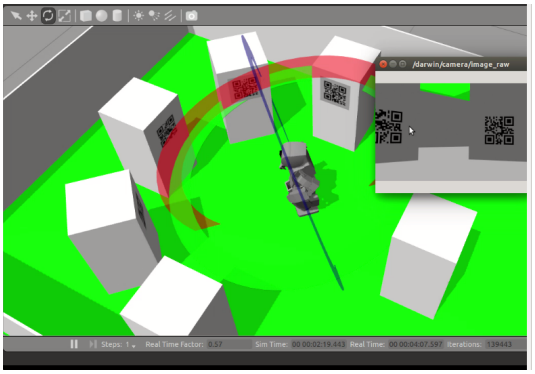
Pel que fa el moviment del robot, ja s'observa en les figures 6.8 com el punt en què el robot inicia i acaba és semblant, això és degut a que s'ha mogut en ajudar-lo i, per tant no s'aprecia el moviment el·liptic.

6.3.3 Resultats del robot real:

En aquest cas, les diferents proves s'executen amb diferents temps. Aquest depèn si les causes adverses són favorables en la detecció del codi QR o no. Les raons dels casos anteriors 6.1 y 6.2 es



(a) Inici



(b) Fi

Figura 6.8: Posició inicial i final en la realització de la tercera prova del robot real.

La taula de resultats de la tercera prova és la següent:

Concepte	Prova1	Prova2	Prova3
Temps Simulació [min]	5.12	1.94	1.02
Nombre de vegades que perd el codi QR [45R]	0	0	0
Nombre de vegades que perd el codi QR [135L]	26	4	9
Nombre de vegades que perd el codi QR [90L]	5	-	14
Nombre de vegades que perd el codi QR [FINISH]	-	-	-
Ha detectat i seguit el codi QR de prova?	No	Si	No
Ha realitzat la prova de forma autonoma?	No	Si	No

Taula 6.6: Resultats de la tercera prova amb el robot real.

mantenen, el robot continua caminant malament, fent que s'aproximi molt als codis QR's i costi més la seva descodificador.

En aquest cas es detecta que el robot no camina gaire bé cap a l'esquerra, ja sigui per raons del balanceig com per raons de la implementació, però condiciona al moviment fins al punt de poder perdre el codi QR si aquest es posiciona molt aprop d'ell.

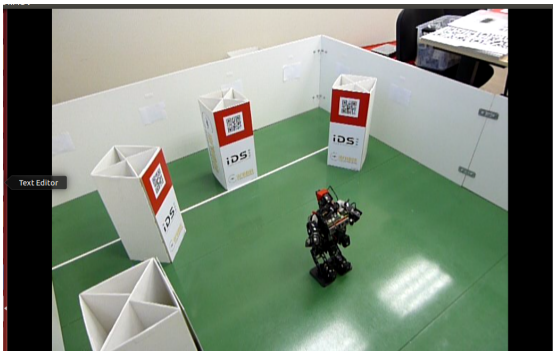
La diferència entre simulació i realitat, és que en la simulació mai es perd el codi QR, per això mai es comprova com actua l'algoritme implementat en cas de pèrdua del codi QR, tal com passa amb el robot real.

6.3.4 Com arreglar els errors:

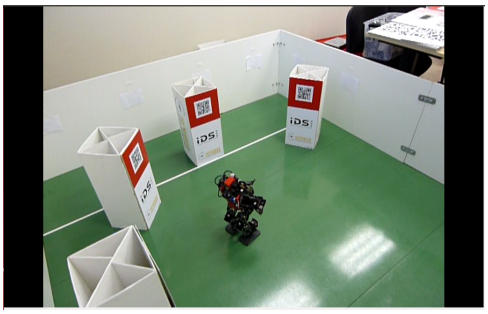
El fet de què el robot camini malament, i, que aquest giri pitjor cap a l'esquerra que cap a la dreta es corrobora en aquesta prova.

Es pot arreglar provant el robot sense la càmera i amb la càmera, posant-la a diferents punts del cap per veure si aquesta li afecta en el balanceig o bé d'un altre lloc.

També, hi ha un problema en el codi, en què en algun estat no s'ha calculat bé l'angle absolut i



(a) Inici



(b) Fi

Figura 6.9: Posició inicial i final en la realització de la tercera prova del robot real.

quan aquest comença la cerca, el servomotor es situa en una posició absoluta molt diferent a la que estava i, per tant es perd i la cerca tarda molt més temps en trobar el que estava buscant. Això s'ha de modificar al codi i verificar-ho.

.

Capítol 7

Anàlisi econòmic

En aquest capítol es detalla el pressupost associat al robot darwinOP, la càmera, les connexions pertinents i el desenvolupament de l'algoritme de la prova de visió. Degut a què el projecte es basa en la implementació d'una nova funcionalitat pel DarwinOP [7], els costos associats no tenen en compte una possible comercialització. Per aquest motiu no s'ha realitzat un anàlisi de viabilitat econòmica.

7.1 Costos

Durant els processos de disseny, construcció i desenvolupament descrits a la memòria, es va fer un seguiment de les despeses. Els costos associats es classifiquen en costos dels materials utilitzats i costos d'enginyeria, els quals es troben detallats en les taules 7.1 i 7.2, respectivament. El cost de l'hora d'enginyer s'ha establert en 12 €/hora. Els càlculs del cost de la peça impresa, s'ha estimat per la grandària i les mesures.

Taula amb els costos del material és la següent:

Component	Cost(€)
Robot darwinOP	8000
Bateria nova	17,31
Peça imprimida	5
Cable mini	2,05
TOTAL	**8024.36**

Taula 7.1: Costos del Material

Cal especificar que les hores de desenvolupament de l'algoritme de la taula **??** referència al desenvolupament dels programes, a la implementació d'aquests dintre del robot, i a l'ajust de

Taula amb els costos de l'enginyeria és la següent:

Concepte	Hores	Cost(€)
Disseny	5	60
Desenvolupament del algoritme	800	9600
TOTAL		**9660**

Taula 7.2: Costos d'Enginyeria

paràmetres per tal d'optimitzar l'algoritme, conjuntament. Sumant el cost total de les dos taules s'obté que el cost del projecte és de 17624.36 [€].

Al inici del projecte ja es disposava de tots els materials de la taula 7.2. Per tant, el cost real del projecte, sense valorar aquests components, és de 24150 [€]. .

Capítol 8

Impacte mediambiental

En aquest capítol es fa un estudi de l'impacte mediambiental associat a la realització del projecte. S'han de distingir tres etapes diferents en les que hi ha la possibilitat d'emetre residus al medi ambient: el procés de fabricació i muntatge, la vida útil del robot i els elements utilitzats, i el final de la vida útil d'aquests. Per tant, s'ha dividit l'estudi en capítols més petits per remarcar cada etapa:

8.1 Fabricació i muntatge:

En aquesta etapa es poden distingir dos grups de components diferents. El primer, el dels materials modificats per fabricar el camp de realització de la prova. I l'altre, el dels components que es van adquirir i que s'utilitzen sense cap mena de modificació, com per exemple l'ordinador, la impressora 3D. Els elements del primer grup, han estat tots els cartons que contenien els codis QR, els codis QRs i el terra fet de **??**. Tant en els residus emesos en la fabricació dels elements del primer i el segon grup, els diferents embalatges, s'han de reciclar convenientment, ja que aquest es consideren residus municipals, d'acord amb l'article 3 del Decret Legislatiu 1/2009, de 21 de juliol. El fet del material de la impressora 3D, es fa càrrec les empreses distribuïdores.

8.2 Vida útil

Tant el robot com l'estació de càrrega no generen residus al llarg de la seva vida útil, ja que cap dels seus components necessita un manteniment. Per tant, només es generen residus en cas de que algun dels components es faci malbé, ja sigui per un deteriorament causat per la seva utilització o per un mal ús d'aquests. Cal tenir present que per utilitzar el robot i l'estació es necessita energia

elèctrica. Degut al fet que la producció d'aquesta energia provoca emissions al medi ambient cal fer un ús responsable d'aquests.

8.3 Final de la vida útil

Un cop arribat el final de la vida Útil del robot, aquest ha de ser desmuntat i reciclat. La major part de la seva estructura és de plàstic, el qual s'ha de dipositar en el contenidor pertinent. A més a més, els cargols i les parts metàl·liques dels actuadors són d'acer, així que tots els components d'aquests materials són fàcilment desmuntables classificables i reciclables portant-los a forns. Tot el cablejat del robot s'ha de portar a la deixalleria, per tal de separar correctament el coure del recobriment de plàstic. Els components electrònics com la controladora, el divisor de tensió o la càmera web, també s'han de dipositar a la deixalleria. Les bateries de liti s'han de dipositar en el contenidor pertinent, ja que son un material molt contaminant. La pantalla de cartró s'ha de dipositar al contenidor blau i la fusta s'ha de portar a un punt verd. I els codis QR's s'han de dipositar al contenidor groc. .

Capítol 9

Conclusions

L'objectiu general del projecte era que el robot hominoide darwinOP realitzés la prova de visió del concurs de forma autònoma, utilitzant un nou programari bàsic més complex.

Primerament, s'ha efectuat l'estudi de les eines necessàries del nou programari bàsic, utilitzades en la implementació de la prova. Seguidament, amb el nou suport creat, aconseguint que la càmera estigui paral·lela als codis QR , s'obté una bona detecció i descodificació dels codis.
Finalment, amb els coneixements del funcionament del robot, darwinOP s'ha creat l'algoritme necessari per a que el robot realitzi la prova de visió amb èxit. Aquesta prova, s'ha provat amb el robot simulat i amb el robot real.

Els resultats obtinguts han estat satisfactoris.

El disseny de la peça ha estat realitzada satisfactòriament, aquesta es fixa perfectament amb el cos i, fa que la càmera estigui paral·lela als codis QR's.

Per una banda, en la prova de simulació, el robot no perdia mai el codi QR, degut a que el seu moviment era més estable. Però, corrobora que la forma de caminar segueix una trajectòria el·líptica, fet que moltes vegades, el robot en caminar i desplaçar-se del centre del camp, aquest es situï molt aprop del codi QR i no el pot detectar. La simulació ha estat l'eina bàsica del projecte ja que ha permès verificar de forma ràpida l'algoritme implementat i, descartar errors.

Per l'altra banda, en la prova amb el robot real, el robot necessita molta més cura. Aquest es perd cada vegada que segueix el codi QR i, es tornava a situar després. Igual que en la prova de simulació, el moviment del robot també es el·líptic, es desplaça i, a vegades, s'aproxima tant al codi QR que no el detecta. Per tant, la prova amb la simulació és més efectiva, ja que no es perd.

També, els grans moviments del cap, provoquen que l'algoritme del robot no es compleixi amb èxit, degut a les restriccions físiques dels servomotors i, que algun cable del hardware disminueixi

la visibilitat, ja que no deixen fer l'algoritme de cerca am bones condicions. Per tant, l'algoritme de cerca funciona perfectament en alguns casos, però en alguns altres falla. En alguns s'ha optat per ajudar al robot; en alguns casos després ha acabat la prova satisfactòriament, però, en altres no.

Finalment, es dóna per vàlid el disseny de l'algoritme implementat, amb alguns aspectes a millorar.Totes les màquines d'estat proposades han estat validades amb èxit en l'execució de l'algoritme, encara que, la de cerca, requereix una millora.

Des d'un punt de vista econòmic i d'impacte ambiental, no es considera cap inconvenient a la viabilitat de la implementació de la prova de visió.

Tot així, cal considerar que aquests anàlisis s'han fet basant-se en els recursos utilitzats en el present projecte, i és necessari que els projectes que es realitzin a posteriori, per a realitzar una altra prova del concurs, facin un anàlisi específic i exhaustiu de les repercussions ambientals i econòmiques.

Amb tot això, s'ha aconseguit establir una base i una guia a partir de la qual es pugui seguir treballant per tal d'optimitzar el procés de visió, i per poder crear nous algoritmes que puguin desenvolupar altres proves.

9.1 Treball Futur

Després del resultat obtingut, cal millorar-lo en alguns aspectes, aquests són els següents:

- Millorar l'algoritme de cerca, caldria incorporar-li que després de centrar-se amb el codi QR trobat, que aquest tirés endarrere uns passos, fins estar a la mateixa distància equidistant a tots els altres, i poder-lo detectar millor.
- Caldria millorar la forma de moure's el robot real, es creu que aquest pot tenir canvis quan el cap no esta al seu lloc, ja que al variar la distribució del pes, pot fer que desequilibri el robot, i el balanceig funcioni d'una fent que aquest camini cap a un altre costat.
- Arreglar el problema de què el robot giri millor a la dreta que a l'esquerra. Tant a nivell de *hardware* com de *software*
- Crear un algoritme de planificació, per realitzar la prova dels obstacles del CEABOT [4].

Capítol 10

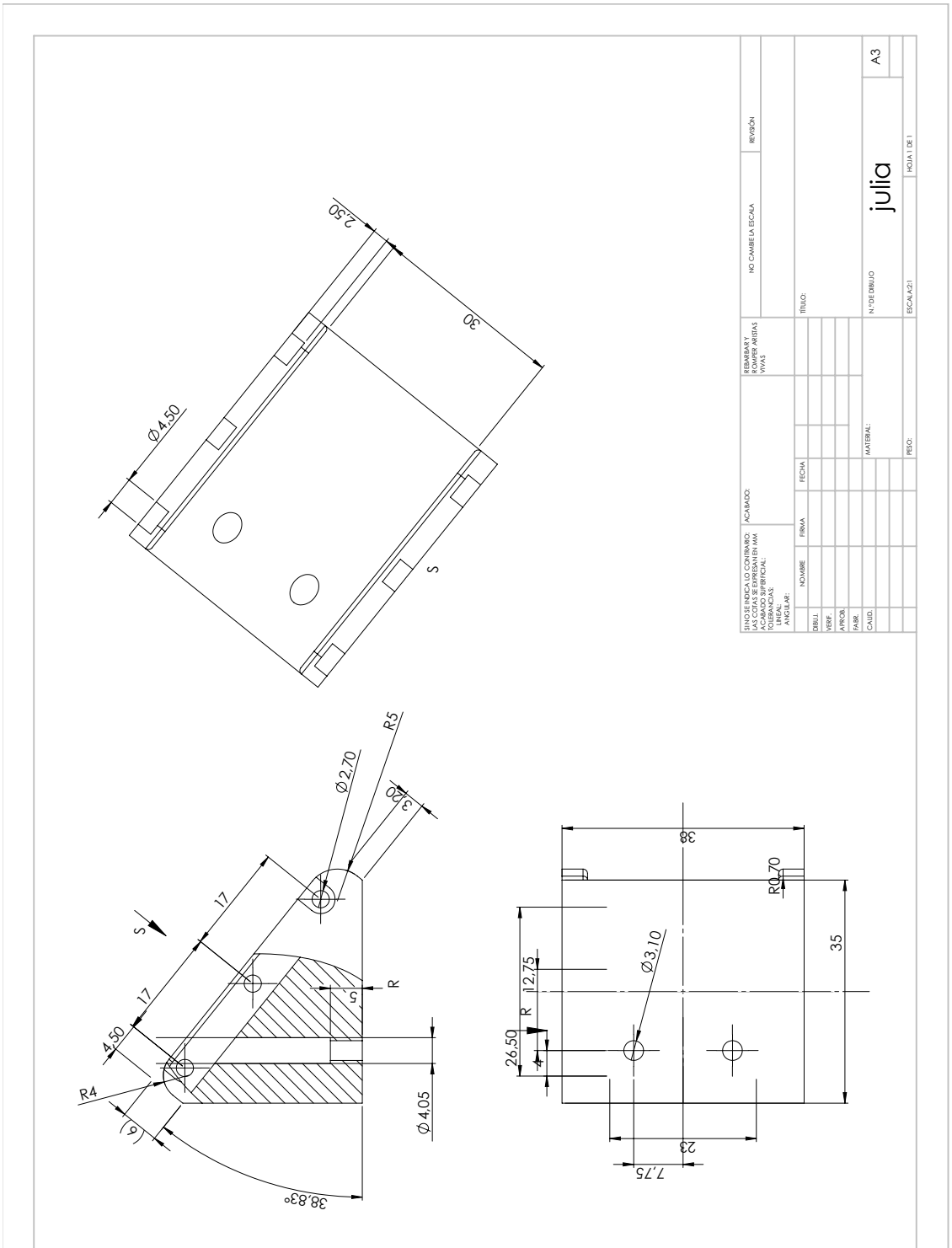
AGRAÏMENTS

Primerament, vull agrair la col·laboració de totes les persones de l'IRI que m'han acompanyat al llarg de tot el projecte, sobretot al Sergi Hernández, al Guillem Alenyà i al Gerard Canal, sense vosaltres, aquest projecte no s'hagués.

Seguidament, volia agrair al meu estimat Victor, per totes aquelles hores invertides en aquest projecte, per la gran paciència i motivació donada. A la meva família, pel suport i la confiança i, als meu germà, amics i companys, per aquest gran camí compartit amb il·lusió i desil·lusió que hem seguit i que ha arribat a la seva fi.

Apèndix A

Annex I: Plànol de la peça del cap



Bibliografia

[1] Anònim. add action server & client. http://wiki.iri.upc.edu/index.php/LabRobotica_ROS_Scripts_Add_Action_Server_Client, August 2016.

[2] Anònim. add server & client. http://wiki.iri.upc.edu/index.php/LabRobotica_ROS_Scripts_Add_Server_Client, August 2016.

[3] Anònim. Callback. http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning, Agost 2016.

[4] Anònim. Ceabot. http://www.ceautomatica.es/sites/default/files/upload/10/CEABOT/index.htm, August 2016.

[5] Anònim. Cm730. http://support.robotis.com/en/product/darwin-op/references/reference/hardware_specifications/electronics/sub_controller_(cm-730).htmt, Agost 2016.

[6] Anònim. codiqr. https://es.wikipedia.org/wiki/C%C3%B3digo_QR, August 2016.

[7] Anònim. Darwinop. http://www.robotis.com/xe/darwin_en, August 2016.

[8] Anònim. dynamixels mx-28. http://support.robotis.com/en/product/dynamixel/mx_series/mx-28.htm, August 2016.

[9] Anònim. Estructura del projecte. http://wiki.iri.upc.edu/index.php/Software_page#Project_structure, Agost 2016.

[10] Anònim. Fitpc. http://www.fit-pc.com/web/products/fit-pc2/, Agost 2016.

[11] Anònim. Generar una estructura. http://wiki.iri.upc.edu/index.php/Create_a_new_development_project#Generate_the_project_structure, Agost 2016.

[12] Anònim. geometry_msgs/twist monthsage. http://docs.ros.org/kinetic/api/geometry_msgs/html/msg/Twist.html, August 2016.

[13] Anònim. Hardware. https://es.wikipedia.org/wiki/Servomotor, August 2016.

[14] Anònim. Ids. https://es.ids-imaging.com/store/produkte/kameras/usb-2-0-kameras/ueye-xs/show/all.html, August 2016.

[15] Anònim. joint_trajectory_action. http://wiki.ros.org/joint_trajectory_actionl, August 2016.

[16] Anònim. Midldleware. https://es.wikipedia.org/wiki/Middleware, Març 2016.

[17] Anònim. Ros/actionlib. http://wiki.ros.org/actionlib, August 2016.

[18] Anònim. Ros/introduction. http://wiki.ros.org/ROS/Introductionl, August 2016.

[19] Anònim. Ros/master. http://wiki.ros.org/Master, August 2016.

[20] Anònim. Ros/nodes. http://wiki.ros.org/Nodes, August 2016.

[21] Anònim. Ros/parameterserver. http://wiki.ros.org/Parameter%20Server, August 2016.

[22] Anònim. Ros/topics. http://wiki.ros.org/Topics, August 2016.

[23] Anònim. Script add publisher & subscriber. http://wiki.iri.upc.edu/index.php/LabRobotica_ROS_Scripts_Add_Publisher_Subscriber, August 2016.

[24] Anònim. Script createpackage. http://wiki.iri.upc.edu/index.php/LabRobotica_ROS_Scripts_Create_Package, Agost 2016.

[25] Anònim. sensor_msgs/jointstate monthsage. http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html, August 2016.

[26] Anònim. trajectory_msgs/jointtrajectorypoint monthsage. http://docs.ros.org/kinetic/api/trajectory_msgs/html/msg/JointTrajectoryPoint.html, August 2016.

[27] cplusplus.com. because cpp. http://www.cplusplus.com/doc/tutorial/, February 2016.

[28] DarthGandalf. Why google c++ testing framework? https://github.com/google/googletest/blob/master/googletest/docs/Primer.md, Agost 2016.

[29] davetcoleman. Simulaodor gazebo. http://wiki.ros.org/simulator_gazebo?distro=groovy, Juny 2016.

[30] Apache Software Foundation. Subversion. https://subversion.apache.org/, Juny 2016.

[31] Open Source Robotics Foundation. Driver base. http://wiki.ros.org/driver_base, Agost 2016.

[32] IRI. Iri. http://www.iri.upc.edu/, August 2016.

[33] Júlia Marsal. iri_vision_prove. https://juliamp@bitbucket.org/juliamp/iri_vision_prove.git, Agost 2016.

[34] MartinGuenther. Joint_trajectory_action. http://wiki.ros.org/joint_trajectory_action, August 2016.

[35] ROBOTIS. Bioloid. http://en.robotis.com/index/, Febrer 2015.

[36] Robotis. Communicació 1.0. http://support.robotis.com/en/product/dynamixel/dxl_communication.htm, Juny 2016.

[37] Robotis. Communicació 2.0. http://support.robotis.com/en/product/dynamixel_pro/communication.htm/, Juny 2016.

[38] ROBOTIS. Servei tècnic del darwinop. http://support.robotis.com/en/, August 2016.

[39] ROBOTIS. Walking tunner. http://support.robotis.com/en/product/darwin-op/development/tools/walking_tuner.htm, August 2016.

[40] Smiguel. Máquines d'estat. http://creaciodigital.upf.edu/~smiguel/b13maquinaEstados.htm, Juliol 2016.