Universidade de Aveiro Departamento de Engenharia Mecânica
2016

Tiago Ferreira Simões

Integração de ROS-Industrial num robô FANUC para flexibilizar atividades de cooperação

Universidade de Aveiro Departamento de Engenharia Mecânica
2016

Tiago Ferreira Simões

Integração de ROS-Industrial num robô FANUC para flexibilizar atividades de cooperação

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestrado em Engenharia Mecânica, realizada sob orientação científica de Vítor Manuel Ferreira dos Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro.

O júri / The jury

Presidente / President	**Prof. Doutor José Paulo Oliveira Santos** Professor Auxiliar da Universidade de Aveiro
Vogais / Committee	**Prof. Doutor António Paulo Gomes Mendes Moreira** Professor Associado da Universidade do Porto - Faculdade de Engenharia (arguente)
	Prof. Doutor Vítor Manuel Ferreira dos Santos Professor Associado da Universidade de Aveiro (orientador)

Agradecimentos / Acknowledgements

Aos meus pais, irmã e a todos os restantes familiares que sempre me apoiaram e deram todas as condições para alcançar os meus objetivos.
Quero agradecer aos meus amigos que, em todos os momentos, estiveram ao meu lado.
Os meus sinceros agradecimentos ao Professor Vítor Santos pela presença constante, motivação e orientação ao longo do trabalho. Ao Eng. Rui Cancela pela ajuda prestada na resolução de problemas pontuais ligados ao robô. Agradeço também ao João Veloso pela ajuda prestada no algoritmo de jogo.

Palavras-chave API, ROS, ROS-Industrial, MoveIt!, Fanuc LR Mate 200iD, Sensor Kinect

Resumo

Esta dissertação descreve a integração de ROS-Industrial no robô Fanuc LR Mate 200iD do LAR. O ROS-Industrial é uma variante do sistema operativo ROS que permite estender a manipuladores industriais as facilidades tradicionais do ROS já comuns em robôs de laboratório ou para investigação. De todas as bibliotecas de diversos fabricantes contidas no ROS-Industrial, utilizaram-se as referentes ao ROS Fanuc, em particular a biblioteca focada no LR Mate 200iD. Recorreu-se à biblioteca MoveIt! para planear trajetórias dinâmicas, verificar colisões e calcular cinemáticas.

O computador atua como cliente, numa arquitetura cliente-servidor, comunicando com o servidor ROS alojado no controlador do robô industrial. O servidor ROS-Industrial implementado neste trabalho, transmite o estado das juntas do robô e recebe as sucessivas posições das juntas que o robô terá de executar.

Adicionalmente, foi desenvolvida uma aplicação de forma a validar e demonstrar a aplicabilidade da API ROS-Industrial no robô Fanuc. Essa aplicação demonstrativa consiste num jogo de damas, e permite a cooperação entre o robô e um humano. Para levar a cabo essa aplicação de cooperação entre o robô e o operador humano, foi também criado um método automático de calibração entre os sistemas de coordenadas do robô, do tabuleiro de damas e do sensor Kinect usado para a perceção do tabuleiro.

A aplicação desenvolvida permitiu a realização de um jogo de damas entre o operador humano e o robô de uma forma adaptativa em que a posição e orientação do tabuleiro podia variar entre jogadas, o que demonstrou a viabilidade e o sucesso da integração do ROS-industrial no robô Fanuc.

Keywords API, ROS, ROS-Industrial, MoveIt!, Fanuc LR Mate 200iD, Kinect Sensor

Abstract

This dissertation describes the integration of ROS-Industrial in the Fanuc LR 200iRD robot. The ROS-Industrial is a variant of the ROS operating system that allows you to extend the traditional facilities of ROS, already common in laboratory robots or used for research, to industrial manipulators.
From the wide range of libraries related to ROS-Industrial, for this project were selected those related to ROS Fanuc, particularly the one which focus on the LR Mate 200iD, the MoveIt! Library. This one was used to plan dynamic trajectories, check collisions, and calculate kinematics.
The computer acts like a client in a client-server architecture, communicating with the ROS server located in the industrial robot controller. The ROS server implemented in this work transmits the state of the robot joints and receives the successive positions of the joints that the robot will have to execute.
In addition, it was developed an application in order to validate and demonstrate the applicability of the ROS-Industrial API in the Fanuc robot. This demo consists of a game of checkers and allows cooperation between the robot and a human being. To carry out this co-operative relationship, it was also created an automatic method of calibration among the robot coordinate systems, the checkerboard and the Kinect sensor, used for the perception of the board.
This application allowed a game of checkers between a human operator and the robot to be played in an adaptive way. The position and the guiding of the board could vary between plays, which proved the viability and the successful integration of ROS-industrial in the Fanuc robot.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Acrónimos e Siglas

ACM *Allowed Collision Matrix*

API *Application Programming Interface*

CAD *Computer Aided Design*

CAM *Computer Aided Manufacturing*

FPS *Frames Per Second*

FTP *File Transfer Protocol*

HTTP *Hypertext Transfer Protocol*

I/O *Inputs/Outputs*

IEEE *Institute of Electrical and Electronics Engineers*

IFR *International Federation of Robotics*

IIC *Intermodalics Intelligent Controller*

IP *Internet Protocol*

IR *Infrared*

KDL *Kinematics and Dynamics Library*

LAR Laboratório de Automação e Robótica

OMPL *Open Motion Planning Library*

OROCOS *Open RObot Control Software*

OpenCV *Open Source Computer Vision Library*

OpenNI *Open Natural Interaction*

PCL *Point Cloud Library*

RANSAC *Random Sample Consensus*

ROS *Robot Operating System*

SRDF *Semantic Robot Description Format*

TCP *Tool Center Point*

TF *Transform Frames*

URDF *Unified Robot Description Format*

USM *User Socket Messaging*Infrared

Capítulo 1

Introdução

Os avanços na área da robótica requerem robôs que procedam mediante a informação do seu meio de forma a serem integrados em diversas aplicações. No campo da cooperação entre robôs, surgem de imediato as questões de atuação simultânea ou sequencial que apresentam problemas e desafios específicos. Distinguem-se duas grandes situações que são a cooperação com e sem a monitorização do contacto entre os intervenientes. Neste trabalho utilizam-se soluções sem contacto entre os agentes ativos.

O conhecimento preciso da posição geométrica dos agentes ativos, tais como os robôs ou outros, permite construir aplicações de cooperação. Porém, quando esse conhecimento não existe, é necessário determinar o posicionamento relativo dos robôs para poderem cooperar. Uma solução possível recorre à perceção do mundo externo mediante um sensor adequado, tal como o sensor Kinect, e uma aplicação computacional.

A construção de uma aplicação computacional deve, não só permitir determinar as posições precisas dos agentes em relação a um referencial comum, mas também possibilitar ao programador controlar e comunicar com um robô sem um profundo conhecimento de aspetos funcionais. Nesta linha, pretende-se implementar uma solução que facilite a comunicação homem-máquina.

1.1 Enquadramento

Verifica-se que a implementação de robôs industriais é um mercado em expansão. Segundo a *International Federation of Robotics* (IFR)[1], em 2014, a totalidade de vendas de robôs industriais atingiu as 229.000 unidades em todo o mundo.

Estima-se que a venda de robôs industriais (figura 1.1) atinja as 400.000 unidades em 2018, 48.000 na América, 275.000 na Ásia/Austrália, 66.00 na Europa e as restantes não especificadas por cidades. A crescente aposta nestes equipamentos deve-se, essencialmente, a avanços tecnológicos que permitem a interligação dos equipamentos fabris (industria 4.0), a simplificação no uso de robôs e a colaboração entre humanos e robôs.

O controlo e a programação de muitos destes robôs nem sempre é um processo simples, sendo muitas das vezes realizado em programação de baixo nível. Conhecimentos tecnológicos da interação entre os diferentes componentes do robô e de uma célula de trabalho, são também necessários. Evidencia-se também o facto da programação ser totalmente dirigida para o equipamento específico, isto é, alterando-se o modelo do robô usado o programa sofre profundas modificações, implicando novos tempos de aprendizagem.

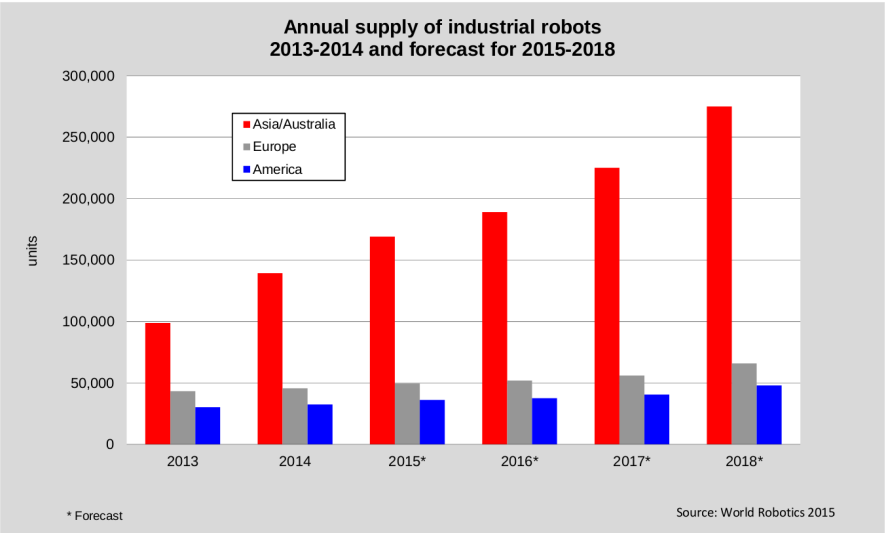


Figura 1.1: Histórico e projeção de vendas de robôs industriais.[1]

De forma a simplificar a programação, utiliza-se uma *Application Programming Interface* (API). API permite utilizar uma linguagem mais próxima da especificação da lógica que traduz o comportamento do sistema, abstraindo o utilizador de detalhes.

Este tema de trabalho surgiu da necessidade de melhorar a interface e a comunicação entre os utilizadores e manipuladores industriais, particularmente os robôs industriais presentes no Laboratório de Automação e Robótica (LAR) da Universidade de Aveiro. Anteriormente a este trabalho, o controlo e a comunicação com os robôs eram assegurados pelo robCOMM [2]. Neste trabalho de dissertação, o autor desenvolveu uma aplicação servidora que disponibilizava um conjunto de serviços a dispositivos remotos, tais como um computador. O servidor robCOMM encontrava-se alojado num controlador de um robô industrial e disponibilizava serviços de movimento, manipulação de dados, cálculos de cinemáticas e execução de programas.

1.2 Objetivos

Com a presente dissertação pretende-se atingir os principais objetivos:

- Desenvolver ou adaptar uma API que permita integrar o robô Fanuc do LAR em aplicações mais complexas de manipulação e cooperação em ambiente ROS;
- Desenvolver uma aplicação que, mediante a perceção do espaço tridimensional, determine as transformações geométricas que relacionem dois ou mais sistemas, como manipuladores e outros agentes;
- Desenvolver uma aplicação ilustrativa da funcionalidade da API como, por exemplo, uma tarefa de cooperação sequencial entre dois robôs ou entre um humano e um robô. Um exemplo será um jogo de tabuleiro num espaço inicialmente desconhecido para o robô;

1.3 Estado da arte

1.3.1 Programação de robôs

Segundo Lozano Perez [3], a programação de robôs é dividida em três categorias: sistemas de guiamento, programação ao nível do robô e da tarefa.

A programação por guiamento é muito simples de implementar. Consiste em levar o robô a uma sequência de posições onde, para cada posição, são guardadas as configurações das juntas. Usualmente, para incrementar as juntas do robô, recorre-se à consola onde se especificam dois tipos de movimentos: no espaço de juntas ou no espaço cartesiano. No espaço de juntas executa-se o movimento independente de cada junta. No espaço cartesiano o movimento pode ser feito relativamente ao sistema de coordenadas da base do robô, ao do *end-effector* ou mesmo a um sistema de coordenadas específico. A principal limitação desta técnica prende-se com a impossibilidade de movimentações condicionadas por informação proveniente de sensores externos.

A programação ao nível do robô vem solucionar a principal limitação da programação por guiamento. Esta forma de programação é a mais utilizada. Concretizam-se tarefas mais elaboradas, onde existe incerteza na posição de um dado objeto, condicionando os movimentos do robô. Uma vez que este método de programação utiliza linguagens de alto nível, tornam-se essenciais grandes competências e formação de forma a desenvolver estratégias para condicionar movimentos.

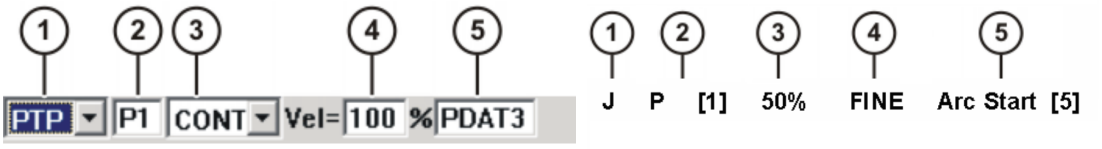
Ao contrário da programação ao nível do robô, a programação ao nível da tarefa define os objetivos para o posicionamento de objetos. O encaixe de um pino guia num furo, por exemplo, necessitará de um conjunto de sequências de movimentos bem específicas do robô para poder inserir. Esta forma de programação pretende uma completa independência de trajetórias específicas a descrever, cinemáticas e geometria do robô.

Os robôs industriais são programados recorrendo a um dos dois métodos possíveis: *online* (utilizando o robô no posto de trabalho) e *offline* (utilizando *software* que não dependa da conexão em tempo real ao robô).

1.3.1.1 Programação *online*

Apesar das linguagens de programação dos diferentes fabricantes parecerem semelhantes tem diferenças na metodologia de programação e na execução dos programas que o compõem. Nas últimas décadas até ao presente, os desenvolvimentos têm-se focado na ferramenta (*end-effector*) indo ao encontro dos conhecimentos num processo de fabrico. [4]

Estas linguagens mantêm as mesmas características face ao seu desenvolvimento original, alterando essencialmente as instruções de comando. O formato de uma linha de instrução é bastante similar alterando essencialmente as instruções de alto nível, focando a aplicação a que se destina. Por exemplo a linguagem TP, proprietária do fabricante KUKA (figura 1.2a), é composta pelo tipo do movimento (1), o nome da posição final (2), o nível de a precisão ao ponto final (3), a velocidade do caminho (4) e o nome para o conjunto de dados de movimento (5). A linguagem TP, proprietária do fabricante Fanuc (figura 1.2b), é composta pelo tipo do movimento (1), os dados de posicionamento (2), a velocidade do caminho (3), o nível de precisão ao ponto final (4) e as opções do movimento (5).



(a) Linha de programação KUKA [5].

(b) Linha de programação Fanuc.

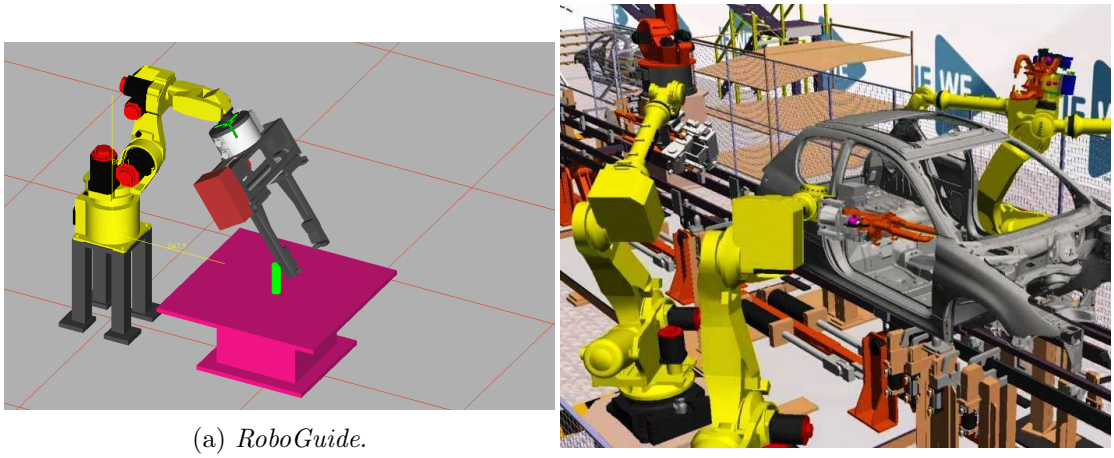
Figura 1.2: Exemplo de linhas de programação dos fabricantes KUKA e Fanuc.

1.3.1.2 Programação *offline*

A programação *offline* oferece um ambiente 3D que permite simular, validar, criar, programar e modelar uma célula de trabalho criando as ferramentas e os equipamentos periféricos dos robôs.

Uma das principais vantagens a destacar é a sua utilização em células complexas de forma a diminuir todo o tempo de desenvolvimento. Em células com dois ou mais robôs, tornava-se bastante dispendiosa e difícil a programação pela consola. Estratégias de simulação tornam-se também uma escolha assertiva para prevenir a ocorrência de erros e estudar melhorias. Em diversos processos, como por exemplo na industria automóvel, a utilização de simuladores impede a destruição de automóveis, o que geraria avultados custos económicos.

Além dos fabricantes de manipuladores disporem de *softwares* para programação *offline* (figura 1.3a), existem aplicações *Computer Aided Design* (CAD) que começaram a incorporar simuladores e um ambiente de programação nos seus *softwares* de forma a simular células complexas (figura 1.3b). Esta incorporação teve como referência o *Computer Aided Manufacturing* (CAM), uma vez que os pós-processadores que geram os programas são semelhantes.



(a) *RoboGuide.*

(b) *Robotics Spot Simulation Engineer.*

Figura 1.3: *Software* para programação *offline* do fabricante Fanuc (RoboGuide) e da Dassault Systèmes (*ROBOTICS SPOT SIMULATION ENGINEER*).

1.3.2 Trabalhos e projetos relacionados

1.3.2.1 Extensão e flexibilização da interface de controlo de um manipulador robótico FANUC

Na dissertação de mestrado de Cancela [2] foi desenvolvido um servidor para o controlador Fanuc RJ3iC, implementando também uma linguagem de suporte à programação e comunicação. O servidor foi criado numa arquitetura cliente-servidor baseado em *Ethernet* e *sockets* TCP/IP. O servidor permite interligar o controlador de um robô industrial a um computador, disponibilizando serviços representativos de tarefas típicas desempenhadas por um manipulador industrial.

A linguagem de suporte à programação desenvolvida permite ao programador, através de uma aplicação cliente, fazer pedidos de serviços de movimento, serviços de manipulação de dados, serviços de cálculo e serviços de execução de programas remotamente. O servidor permite execuções em coordenadas de junta ou cartesiano, cálculo de cinemáticas, acesso a registos e programas, como exemplifica a tabela 1.1.

No entanto, verificou-se que cada instrução precisa de um elevado número de parâmetros de entrada que precisam de ser respeitados. Os valores inseridos e enviados para o servidor precisam ser valores coerentes de modo que o servidor os reconheça e os execute no robô. A necessidade de um elevado número de parâmetros específicos exige que o utilizador tenha bons conhecimentos em robótica. Por exemplo, a instrução `MOVTOCPOS(X,Y,Z,w,p,r,Cfg1,Cfg2,Cfg3,Turn1,Turn2,Turn3)` tem 12 parâmetros de entrada. Os seis primeiros parâmetros definem a posição cartesiana, os seguintes três parâmetros atribuem as redundâncias e os três últimos a rotação em torno dos eixos.

Contempla também uma opção de gestão de erros que serve como validação interna, podendo ou não informar a aplicação cliente da sua origem. A gestão de erros implementada funciona a dois níveis. O primeiro nível verifica a validade dos comandos recebidos pelo servidor e verifica se o serviço está disponível. O segundo nível avalia se o comando recebido pelo servidor é executável.

Tabela 1.1: Lista de instruções da linguagem robCOMM [2].

MOVTOCPOS	Movimento cartesiano
MOVTOJPOS	Movimento em junta
GETCRCPOS	Posição atual do TCP
GETCRJPOS	Configuração atual das juntas
CHECKCPOS	Verifica posições cartesianas
CHECKJPOS	Verifica configurações de junta
GETDIRKIN	Calcula cinemática direta
GETREVKIN	Calcula cinemática inversa
GETREG	Obtem valores de registos
SETREG	Escreve em registos
SETPATH	Compõe um caminho
MOVTHPTH	Move ao longo de um caminho
LISTTPP	Obtem a lista de programas TP
RUNTPP	Executa programas TP
GETDIO	Devolve o estado de I/O digitais
GETAIO	Devolve o estado de I/O analógicas
SETDIO	Atua I/O digitais
SETAIO	Atua I/O analógicas
SETTOOLFRM	Especifica uma Tool frame
SETUSERFRM	Especifica um User frame
SETLSPEED	Especifica velocidade linear
MOTIONSTOP	Para o movimento
STOPSERV	Termina o servidor

1.3.2.2 Projeto *Robotic Blending*

A equipa de investigadores de *Southwest Research Institute*, *The Boeing Company*, *Caterpillar*, *Wolf Robotics* e *TU Delft* desenvolveu e completou uma aplicação de *Scan-N-Plan* [6], focada num projeto técnico, aplicado a um robô industrial.

Entende-se pela tecnologia de *Scan-N-Plan* a ação de planear trajetórias em tempo real, recorrendo a um conjunto de ferramentas que adquiram informação do processo como, por exemplo, uma digitalização 3D. Esta metodologia aplica-se a superfícies que necessitam que sejam eliminados fatores que dão origem a falhas por fadiga. Finalizado este processo, as superfícies ficam preparadas para, por exemplo, uma operação de pintura. *Scan-N-Plan* aplica-se em diversas situações particulares, tais como [6]:

- Não existam modelos CAD disponíveis;
- Incluam peças flexíveis ou deformáveis tais que a pré-programação era impraticável;
- Exijam flexibilidade num processo;

O objetivo deste projeto permite que um operador coloque uma peça que exija um tratamento numa célula de trabalho de um robô. Após a presença da peça o robô gera, automaticamente, um plano para maquinar e executa-o. Com uma simples interface, o operador acompanha o procedimento que inicia pelo *scan* (figura 1.4a), passando pelo planeamento (figura 1.4b), a execução (figura 1.4c) até à inspeção da qualidade da superfície maquinada (figura 1.4d).

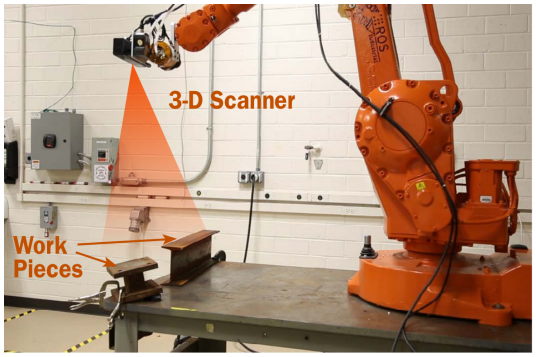
Esta aplicação foi implementada, em dois robôs diferentes, com sucesso nas instalações da Boeing e Caterpillar a 27 de Julho de 2015.

1.3.2.3 Projeto paletização de caixas

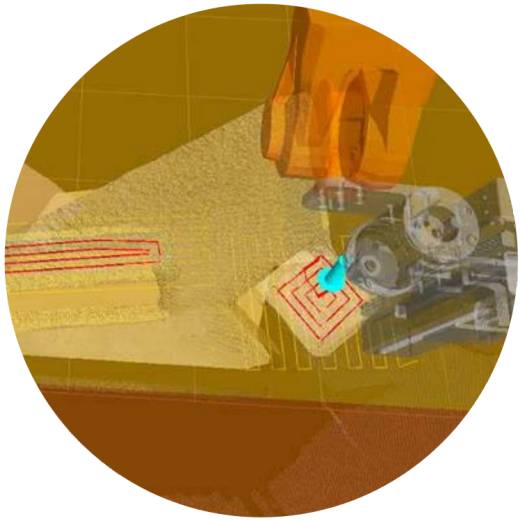
Em dezembro de 2013, a *Altern Mechatronics*, em colaboração com *CSi Palletising Systems*, desenvolveu uma aplicação, baseada em ROS-Industrial, de paletização de caixas usando o robô ABB IRB6640 [7]. Criou-se um ambiente experimental onde se posicionaram duas paletes na zona de trabalho do robô, sendo que uma delas continha três caixas de tamanho desconhecido. O tamanho, a posição e a orientação das caixas foram determinadas com auxílio do sensor Kinect e das bibliotecas da OpenNi e da PCL. Com base nesta informação, o percurso a descrever foi calculado pela biblioteca MoveIt! e em seguida enviado para o controlador do robô que contém o servidor ROS-Industrial da ABB [7].

A aplicação desenvolvida fazia a recolha e o tratamento da nuvem de pontos (figura 1.5a) para detetar as caixas. Após se selecionar a caixa a mover, calcula-se a trajetória que irá transportar a caixa de uma palete para a outra (figura 1.5b). Por fim, o robô executa a trajetória calculada (figura 1.5c) e o processo repete-se novamente (figura 1.5d) até se verificar a ausência de caixas.

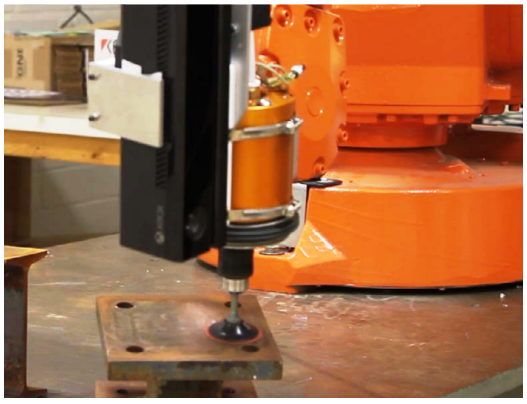
O desenvolvimento *offline* realizou-se por via de uma aplicação desenvolvida pela *Altern Mechatronics*. Esta aplicação permitia a conexão com o *RobotStudio*, *software* disponibilizado pelo fabricante. A aplicação alcançou uma precisão de 1 cm, o que indicava que o sistema de visão necessitava de ser aperfeiçoado. A melhoria passaria por implementar sensores mais precisos como triangulação laser ou câmaras de tempo de voo com algoritmos de filtragem mais avançados.



(a) *Scanning* das superfícies.



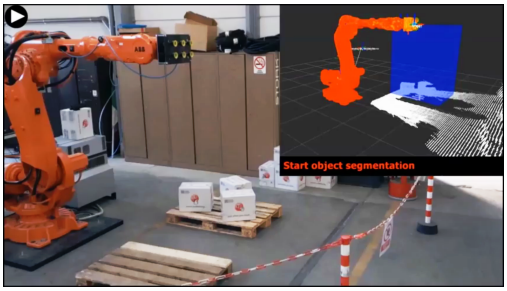
(b) Cálculo e visualização do caminho.



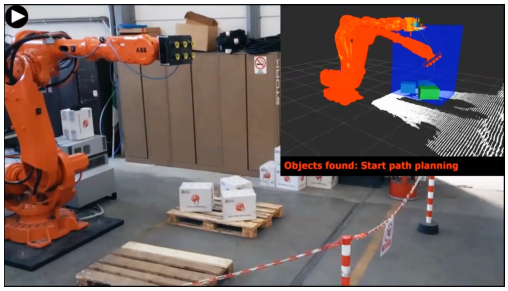
(c) Execução do caminho na superfície.

(d) Verificação da qualidade.

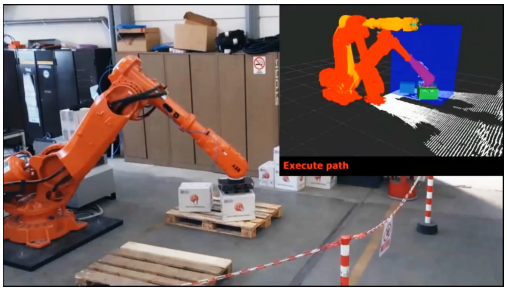
Figura 1.4: Etapas que compõem o projeto *Robotic Blending* [6].



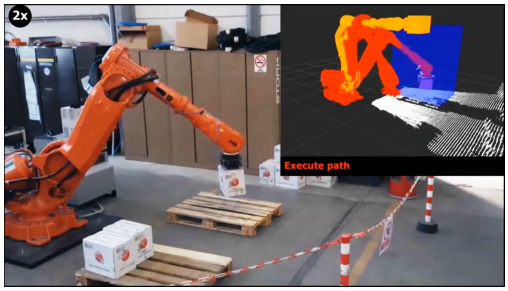
(a) Tratamento da nuvem de pontos.



(b) Seleção do objeto e cálculo da trajetória.



(c) Execução de uma trajetória de *pick*.



(d) Execução de uma trajetória de *place*.

Figura 1.5: Demonstração da aplicação de paletização [7].

Para permitir o controlo e o acionamento do *end-effector*, através dos I/O do robô, adaptou-se o controlador IRC5. Dada a dificuldade na adaptação sugere-se desenvolver uma interface padrão, transversal a todos os robôs, para o controle dos I/O do robô em ambiente ROS-Industrial. Esta interface padrão tornaria mais simples a utilização dos I/O em aplicações [7].

1.3.2.4 Projeto despaletização de caixas

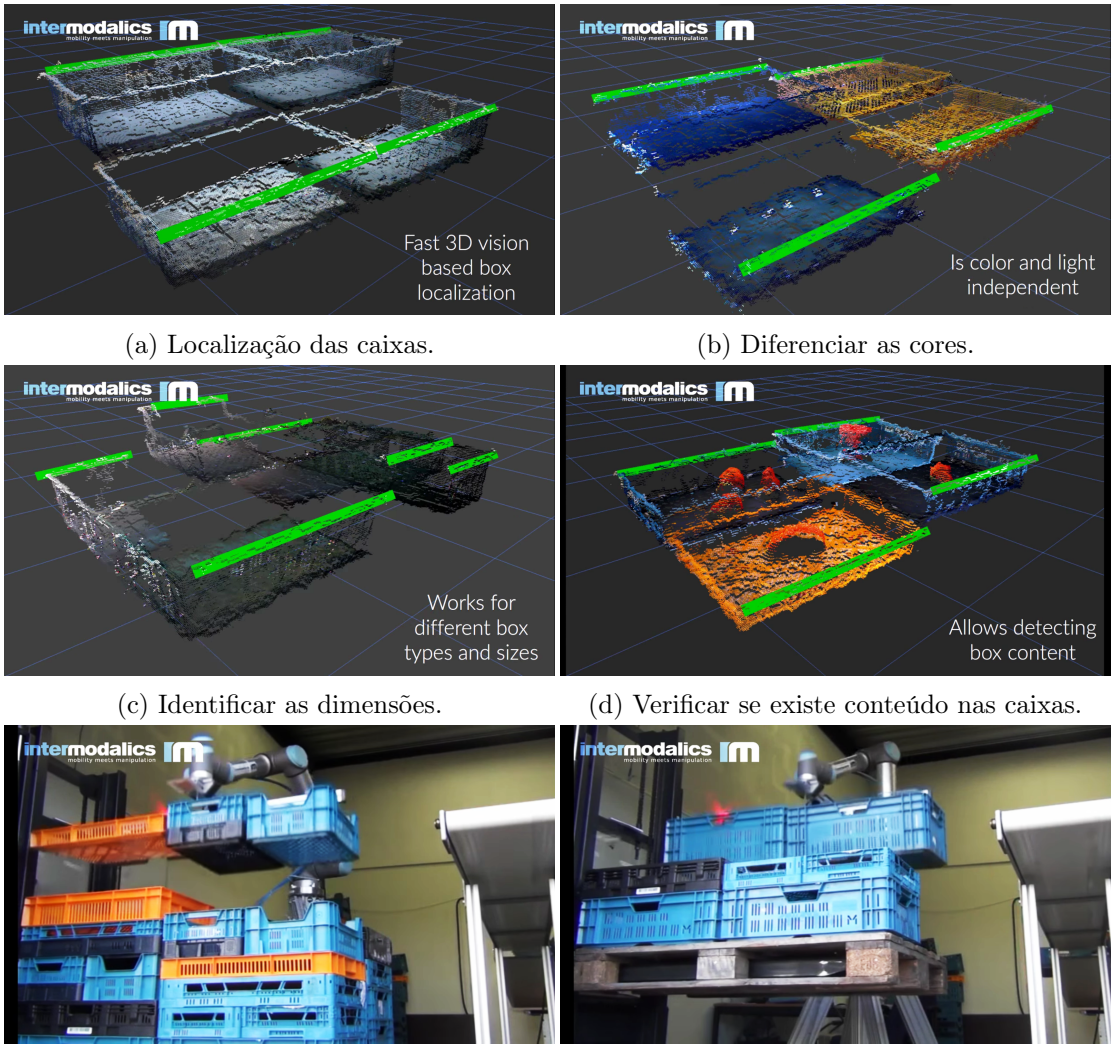
A *Intermodalics* desenvolveu uma solução de despaletização para um cliente [8], cujas necessidades eram de mover, em média, duas mil caixas por hora para um transportador. Adicionalmente, surgiram desafios como as caixas terem cores variáveis e estarem empilhadas de forma aleatória.

A aplicação criada consistiu no robô UR10, da *Universal Robots*, num sensor 3D, num *Intermodalics Intelligent Controller* (IIC) e num elevador que permitia que as caixas estivessem à mesma altura. O IIC é adaptado às necessidades da aplicação e é capaz de fornecer um movimento fiável e seguro para aplicações exigentes de alto nível. O *software* utilizado foi o IIC que utiliza o ROS e a ferramenta *Open RObot Control Software* (OROCOS) como base. OROCOS está totalmente integrado no ROS e é um *software* que contém bibliotecas para controlo avançado, em tempo real, de robôs e máquinas.

Para determinar as posições e orientações das caixas, a *Intermodalics* desenvolveu um conjunto de algoritmos destinados à localização de caixas, baseado na biblioteca PCL. A ferramenta de visualização do ROS (o RViz) verificou-se uma ferramenta indispensável durante todo o desenvolvimento.

O uso da *package* do ROS-Industrial para o robô UR permitiu a simulação dos movimentos do manipulador e do transportador, o que facilitou a implementação de toda a aplicação. Nas figuras 1.6a à 1.6f estão presentes excertos do tratamento da nuvem de

pontos recolhida e da aplicação implementada.



(a) Localização das caixas.

(b) Diferenciar as cores.

(c) Identificar as dimensões.

(d) Verificar se existe conteúdo nas caixas.

(e) *Pick* de caixas de tamanhos diferentes.

(f) *Pick* da última camada.

Figura 1.6: Análise da nuvem de pontos de forma a tornar exequível a despaletização das caixas [8].

1.3.2.5 Observações dos trabalhos e projetos relacionados

A análise de trabalhos e projetos relacionados focou-se em soluções baseadas em API, as quais influenciaram a escolha da API mais adequada para implementar no robô do LAR. De forma a criar aplicações mais complexas que incluam robôs industriais a um baixo custo e sem depender de conhecimentos profundos de robótica, neste trabalho optou-se por explorar os recursos do ROS-Industrial.

Apesar do desenvolvimento do ROS-Industrial ser recente, verifica-se que tem grande aplicabilidade ao nível industrial. Projetos que necessitem de soluções de manipulação, de controle de trajetórias e que procedam mediante a informação recolhida de sensores estão abrangidos pelo ROS-Industrial.

1.4 Estrutura da dissertação

Esta dissertação é composta por sete capítulos. O presente capítulo faz um breve enquadramento e descreve projetos e trabalhos que se enquadram no tema do trabalho. Os restantes capítulos estão organizados da seguinte forma:

- **Estrutura experimental e ferramentas** - Descrição do principal *hardware* e *software* usado neste trabalho;
- **Integração da API ROS-Industrial** - Descrição dos detalhes da API e a sua integração no robô do LAR;
- **Calibrações** - Descrição da metodologia adotada de forma a determinar a posição relativa entre todos os sistemas de coordenadas que compõe a aplicação demonstrativa;
- **Perceção visual** - Análise das metodologias de visão adotadas de forma a recolher informação relevante para a aplicação demonstrativa;
- **Aplicação ilustrativa** - Descrição dos algoritmos e metodologias que compuseram a aplicação ilustrativa;
- **Conclusões e trabalho futuro** - São apresentadas as conclusões do trabalho e as propostas de trabalho futuro;

Capítulo 2

Infraestrutura experimental e ferramentas

Nesta secção estão descritos os principais sistemas de *hardware*, *software* e bibliotecas usados neste trabalho.

2.1 Fanuc LR Mate 200iD

No presente trabalho é utilizado um manipulador Fanuc presente no LAR do Departamento de Engenharia Mecânica da Universidade de Aveiro. O manipulador em causa é compacto e tem uma vasta aplicabilidade em ambiente industrial, proporcionando a execução de diversas operações tais como *pick and place*.

O Fanuc LR Mate 200iD (figura 2.1) é um manipulador composto por 6 juntas, com um alcance máximo de 717 mm, capacidade para suportar uma carga máxima de 7 kg e uma repetibilidade de ±0.02 mm, como indicado na tabela 2.1.

Figura 2.1: Fanuc LR Mate 200iD.

O seu espaço de trabalho, região dentro do qual pode posicionar o *end-effector*, está condicionado não só pelas suas dimensões físicas (figura 2.2) mas também pelos seus limites das juntas.

Tabela 2.1: Especificações Fanuc LR Mate 200iD. [9]

Model		LR Mate 200*i*D
Controlled axes		6 axes (J1, J2, J3, J4, J5, J6)
Reach		717mm
Installation (Note 1)		Floor, Upside-down, Angle mount
Motion range (Maximum speed)	J1 axis	340°/ 360° (option) (450°/s) 5.93 rad/6.28 rad (option) (7.85 rad/s)
	J2 axis	245° (380°/s) 4.28 rad (6.63rad/s)
	J3 axis	420° (520°/s) 7.33 rad (9.08rad/s)
	J4 axis	380° (550°/s) 6.63 rad (9.60 rad/s)
	J5 axis	250° (545°/s) 4.36 rad (9.51 rad/s)
	J6 axis	720° (1000°/s) 12.57 rad (17.45 rad/s)
Max. load capacity at wrist		7kg
Allowable load moment at wrist	J4 axis	16.6 N·m
	J5 axis	16.6 N·m
	J6 axis	9.4 N·m
Allowable load inertia at wrist	J4 axis	0.47 kg·m²
	J5 axis	0.47 kg·m²
	J6 axis	0.15 kg·m²
Repeatability		± 0.02 mm
Mass (Note 2)		25 kg
Installation environment		Ambient temperature : 0~45℃ Ambient humidity : Normally 75%RH or less (No dew nor frost allowed), Short term 95%RH or less (within one month) Vibration : 0.5G or less

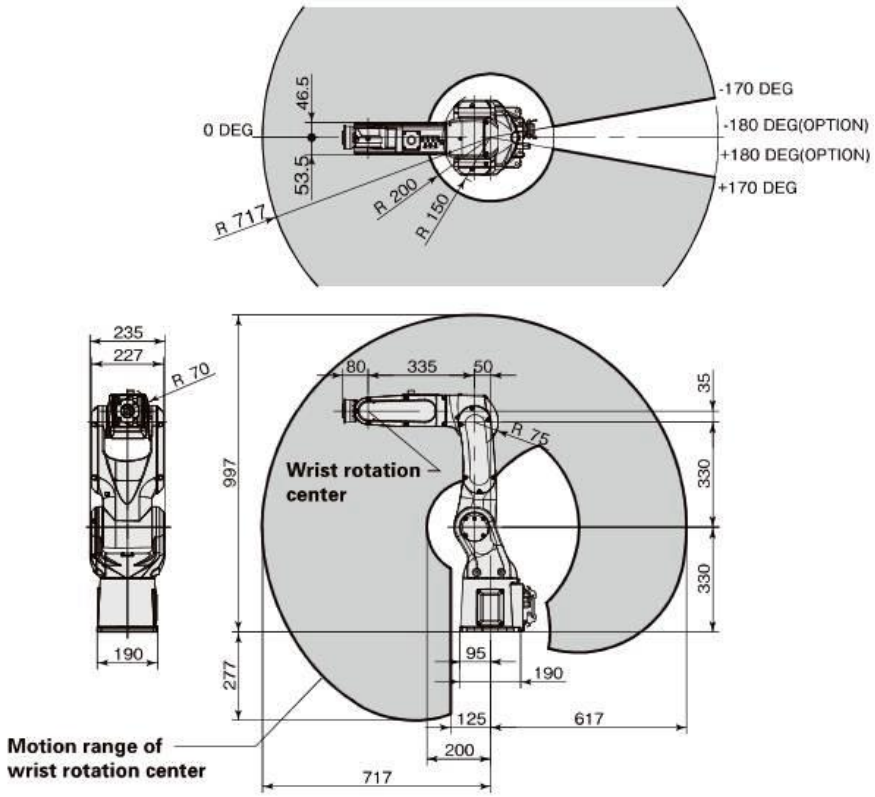


Figura 2.2: Dimensões físicas e espaço de trabalho da unidade mecânica.[9]

2.2 Controlador

O controlador é o elemento fundamental que integra e comanda o *hardware* do manipulador. No presente trabalho o controlador é o R-30iB Mate (figura 2.3), garantindo um elevado nível de desempenho e velocidade de processamento. As principais diferenças que se verifica é ao nível do dimensionamento físico, sendo mais pequeno e silencioso que as versões anteriores.

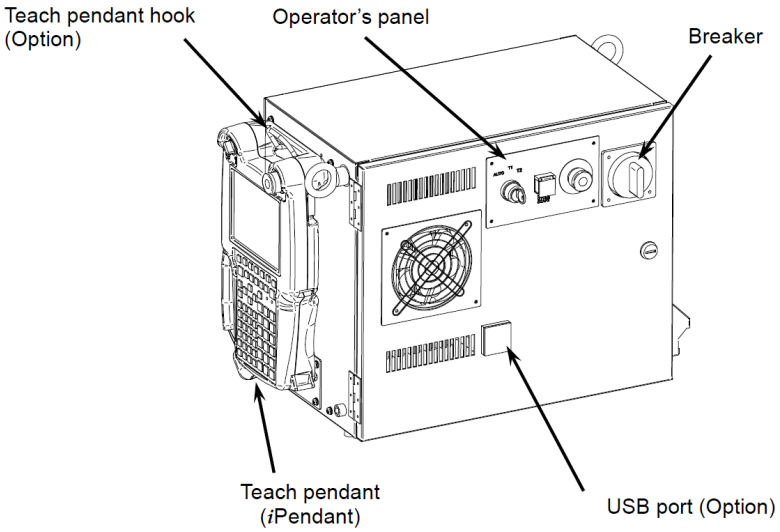


Figura 2.3: Controlador R-30iB Mate.[10]

Para estabelecer a comunicação com o controlador, estão disponíveis diversas conexões para a comunicação com o sistema. Como assinalado na figura 2.4, no presente trabalho foram utilizadas a unidade mecânica, conexão USB, consola de programação, interface *Inputs/Outputs* (I/O), Ethernet e *AC power supply*. De salientar que a referida figura inclui dois tipos de linhas a traço interrompido indicando as ligações mecânicas e as restantes linhas representando as conexões elétricas.

De seguida, é feita uma descrição dos elementos da figura 2.4. Na unidade mecânica está presente a conexão com o *end-effector* que é o responsável pela interação com o meio ambiente. Por outro lado, a pressão pneumática possibilita a execução de uma dada tarefa, como por exemplo, o *pick* de objetos do meio ambiente.

A conexão USB torna possível operações de atualização, *backups* do sistema ou mesmo transferência de ficheiros. A consola é o principal meio de interação com o controlador, possibilitando operações de programação, execução e edição de programas, verificação de erros, estado dos I/O, entre outros.

A interface com I/O é uma excelente opção de controlo dado que é possível aceder, controlar e comunicar com unidades periféricas, como por exemplo sensores e atuadores. Na maioria das redes de campo, a sua conectividade é assegurada por portas série e *Ethernet*. Dadas as vantagens, foi adotada a conexão *Ethernet*. Na secção 2.3 será apresentado e detalhado este meio de comunicação. Por último, surge o AC *power supply* que é o responsável pela alimentação de todo o sistema[10].

O controlador também dispõe de clientes e servidores *Hypertext Transfer Protocol*

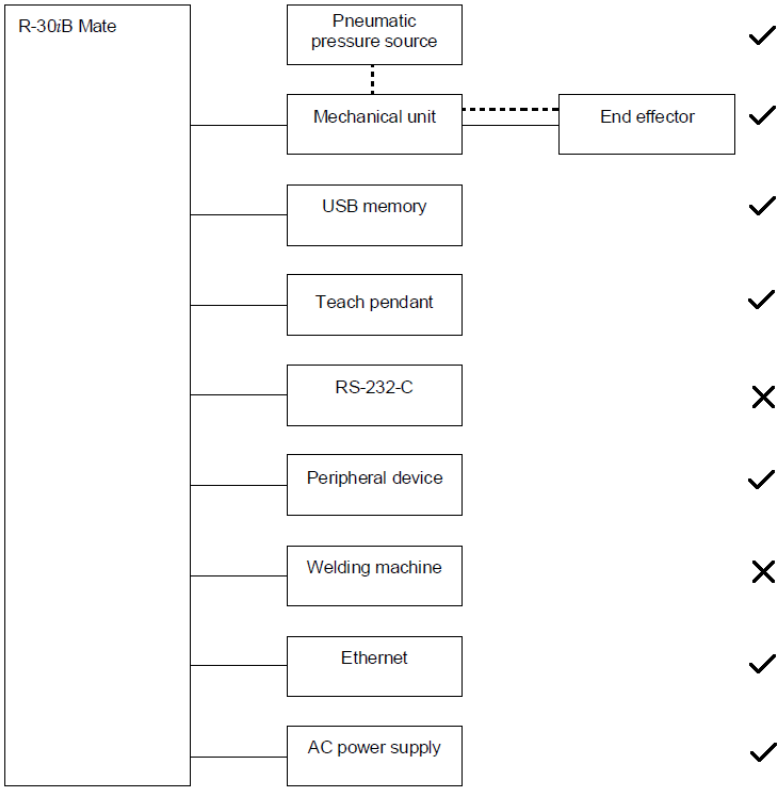


Figura 2.4: Diagrama de blocos das ligações ao controlador, especificando as utilizadas no trabalho.[10]

(HTTP) e *File Transfer Protocol* (FTP) implementados de base. A plataforma focada em HTTP foi criada em [2] possibilitando a monitorização de alguns aspetos do manipulador. Por outro lado, o FTP foi utilizado neste trabalho para a transferência de programas criados por um pós-processador, verificando-se um método rápido e versátil de transferir informação.

2.3 Comunicação

Uma comunicação entre máquinas permite uma melhor compreensão de cada máquina, melhorando não só a sua eficiência mas também uma otimização de vários processos.

A conetividade com controladores industriais poderá ser dividida em três categorias [2]:

- conetividade com equipamentos periféricos, recorrendo a redes de campo e a I/O digitais e analógicos convencionais;
- conetividade com *hardware* de comunicação para transferências de dados ou programas da consola por porta série, USB, slots PCMCIA, etc.;
- comunicação Ethernet, na qual podem ser implementados servidores HTTP e FTP. Pode também servir de meio de comunicação de alto débito com um computador externo;

2.3.1 Ethernet

O sistema Ethernet, proposto pela empresa Xerox e aprovado em 1983 pelo *Institute of Electrical and Electronics Engineers* (IEEE), define que cada equipamento, antes de enviar os seus dados/sinais, deve escutar o meio de transmissão. Quando não houver atividade no meio, deve então enviar os seus dados. Assim, este protocolo permite que diversos equipamentos acedam ao meio de transmissão.

De entre as diversas camadas físicas, a adotada neste trabalho foi o meio de transmissão 10Base-T (figura 2.5). Existem três tipos de diálogos, *simplex*, *half* e *full duplex*. Quando apenas um dos equipamentos se limita a enviar informação e o outro se limita a receber, considera-se que a comunicação é *simplex*. De outra forma, quando os dois equipamentos podem receber e enviar dados, mas não em simultâneo, considera-se que o diálogo é do tipo *half duplex*. Por último, o diálogo *full duplex* é similar ao *half duplex*, diferenciando na possibilidade de dialogar simultaneamente. O diálogo *full duplex* foi o adotado neste trabalho.

2.3.2 TCP/IP

O protocolo *Internet Protocol* (IP), criado pelo departamento de defesa dos Estados Unidos, permite a troca de blocos de dados entre computadores. Cada equipamento tem um único endereço, o que permite encaminhar os blocos de dados entre equipamentos até ao equipamento de destino. O protocolo também possibilita a fragmentação dos dados a enviar (figura 2.6) para que possam ser transmitidas através de redes locais como a *Ethernet*. A mensagem Internet é constituída pelo cabeçalho e pelos dados. Enquanto

Figura 2.5: 10Base-T.

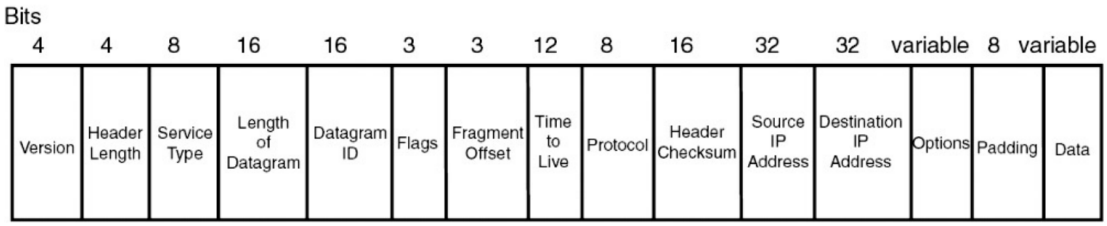


Figura 2.6: Mensagem Internet.

que o cabeçalho contém os endereços de origem e destino das mensagens Internet os dados são as mensagens da camada de transporte. A mensagem IP é enviada para a rede, depois de encapsulada na mensagem da camada lógica, ou diretamente (*MAC-frame*) de uma rede local, habitualmente na mensagem Ethernet.

O protocolo TCP (Transmission Control Protocol), tal como o protocolo IP foi desenvolvido pelo departamento de defesa dos Estados Unidos. Este protocolo fornece um serviço fiável de transferências de dados, apesar dos seus serviços serem prestados pelo protocolo IP, garantindo que não há pacotes perdidos e que são processados na ordem correta.

2.4 Bancada, iluminação e *setup* de jogo

No decorrer do desenvolvimento da aplicação demonstrativa, verificou-se que as condições construtivas atuais (figura 2.7) não garantiam um posicionamento rigoroso do sistema de visão. De forma a que o posicionamento fosse ajustável e garantisse rigor nos dados recolhidos, concebeu-se a estrutura da figura 2.8.

Figura 2.7: Bancada de trabalho original. [11]

Inicialmente, a estrutura era composta por quatro perfis, dois perpendiculares e um paralelo à superfície da mesa de trabalho. Devido a problemas vibratórios gerados durante os movimentos do robô, aumentou-se a rigidez do sistema, aumentando a quantidade de perfis. O posicionamento do sensor era assegurado por um perfil de menores dimensões que unia a estrutura ao suporte do sensor. A fixação da estrutura à mesa fez também com que o sensor ficasse solidário com a posição da mesa de trabalho.

Durante a implementação da visão artificial, verificou-se que a informação recolhida era fortemente influenciada pela interferência da luz solar. Uma vez que o laboratório não é imune às variações de iluminação, implementou-se um ambiente no qual a iluminação pudesse ser, o mais possível, constante. A solução adotada foi o posicionamento de

três projetores de luz fria led 20W na estrutura da bancada. O seu posicionamento foi estudado de forma a diminuir não só a probabilidade de interferência de radiação externa, mas também para que não ocorressem sombras em redor de cada peça, o que influenciaria a deteção por visão. De salientar que os projetores necessitaram de ser adaptados com um acrílico fosco, uma vez que a incidência direta na superfície das peças fazia com que algumas superfícies ficassem espelhadas.

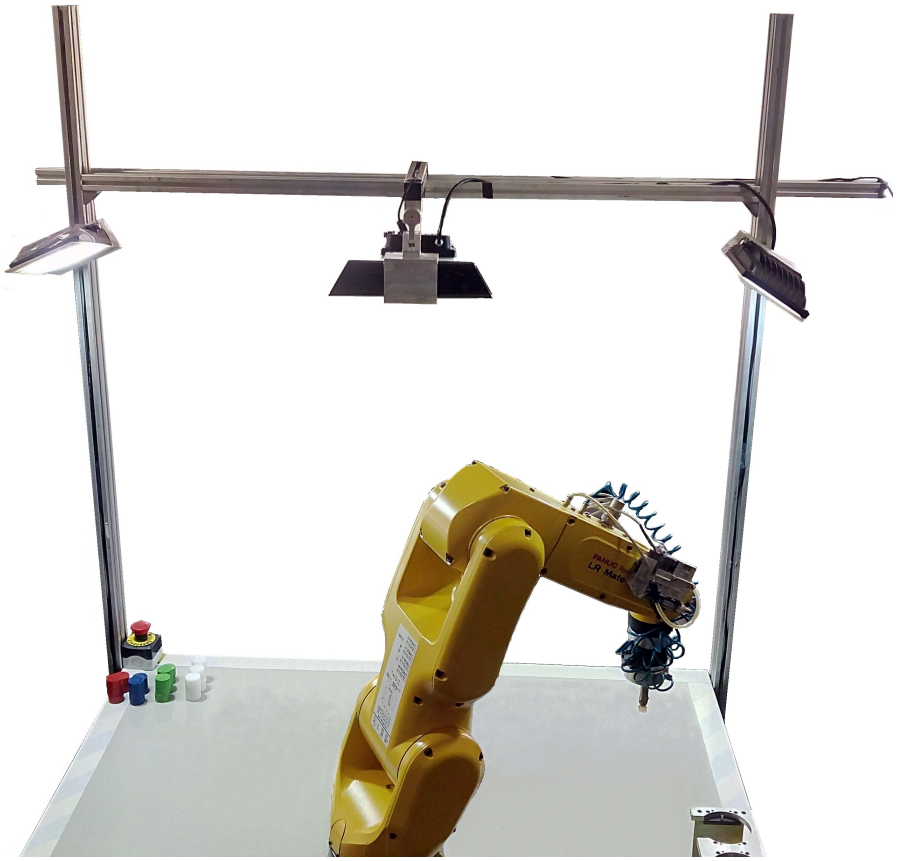
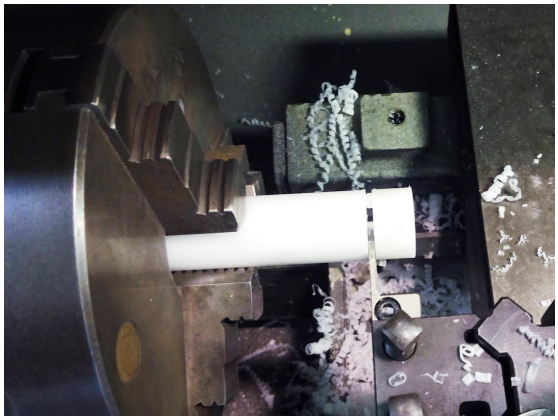


Figura 2.8: Bancada de trabalho composta.

Com o objetivo de criar uma interação entre o manipulador e o utilizador, foi construído um tabuleiro de xadrez bem como as peças de jogo. Uma vez que as peças necessitavam de interagir com a ventosa, com 15 milímetros de diâmetro, e de respeitar a arquitetura do jogo, estas necessitavam de ser redondas, leves e com uma superfície aderente. O nylon foi o material selecionado, uma vez que cumpre todos os requisitos anteriores. Dado que que o nylon se encontrava em forma de varão, maquinou-se o varão (figura 2.9a) para obter as alturas desejadas das peças. O diâmetro, 30 mm, foi selecionado tendo em conta as dimensões do tabuleiro e da ventosa. Inicialmente, testaram-se três alturas diferentes, 10, 15 e 17 mm, de forma a selecionar a altura que garantisse uma boa qualidade dos dados obtidos por parte do sensor Kinect. A altura selecionada foi de 17 mm, (figura 6.11) e foram criadas 36 peças.



(a) Maquinagem das peças.

(b) Peças utilizadas no jogo das damas.

Figura 2.9: Maquinação do varão de nylon e as peças de jogo.

2.5 Sensor Kinect

A Microsoft lançou em Novembro de 2010, utilizando tecnologia da empresa Israelita PrimeSence, o sensor Kinect. Este sensor destinava-se a aplicações lúdicas e de entretenimento, associadas à Xbox 360. Dado o seu baixo custo e a sua fácil integração em ambiente ROS, esta aplicação começou a migrar para a investigação. A versão 1.8 do sensor Kinect necessita de um adaptador para fornecer energia adicional, possibilitando a conexão à porta USB do computador.

Na figura 2.10 são representados os seus constituintes. Estes são divisíveis em 4 grandes grupos: motor, microfones, sensor RGB e um sensor de profundidade, composto por um recetor e emissor de *Infrared* (IR).

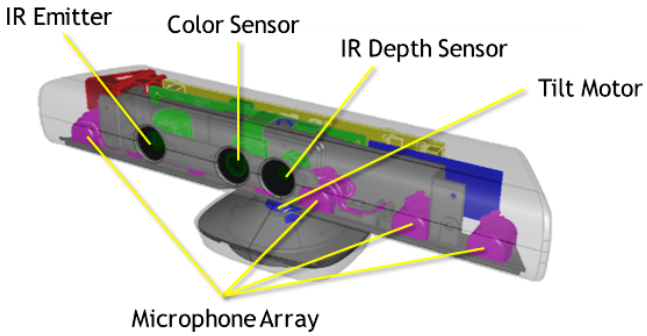


Figura 2.10: Componentes do sensor Kinect.[12]

A posição geométrica dos quatro microfones possibilita a obtenção de uma localização mais precisa do agente que emite o som.

As imagens a cores têm uma resolução de 640x480. O emissor de luz infravermelha e o sensor de profundidade estão diretamente relacionados. Ao ser projetado um padrão de infravermelhos sobre uma região de interesse, torna-se possível analisar a sua reflexão e convertê-la em informação de distância ao sensor. A resolução das imagens de profundidade é de 640x480 com um *frame rate* de 30 *Frames Per Second* (FPS).

Apesar da vasta quantidade de recursos que este equipamento disponibiliza, neste trabalho foram apenas processados os dados do sensor RGB e do sensor de distâncias. Para que o sensor possa ser utilizado por completo em ambiente Ubuntu, recorreu-se à biblioteca *Open Natural Interaction* (OpenNI).

Fundada em 2010 pela PrimeSence, a OpenNI promove e padroniza a interoperabilidade entre o computador e os sensores, como por exemplo, o sensor Kinect. Aquela contém uma variedade de bibliotecas, ferramentas, *packages* e aplicações que estão disponíveis gratuitamente, de forma a que seja possivel recolher e manipular a informação recolhida pelo *hardware* do sensor.[13]

2.6 FreeCAD

O FreeCAD é um modelador 3D genérico de CAD, disponível com licenças GPL e LGPL, ou seja, *open source*. De entre diversas aplicabilidades, o seu uso é preferencialmente direcionado para a área de engenharia mecânica e desenvolvimento do produto. As suas ferramentas são comparadas a grandes *softwares* como CATIA, SolidWorks ou Solid Edge, não tendo tanto foco em animações dinâmicas ou em *Rendering* [14].

Relativamente à sua utilização no trabalho, o FreeCAD foi utilizado na implementação do modelo do *end-effector* (figura 2.11).

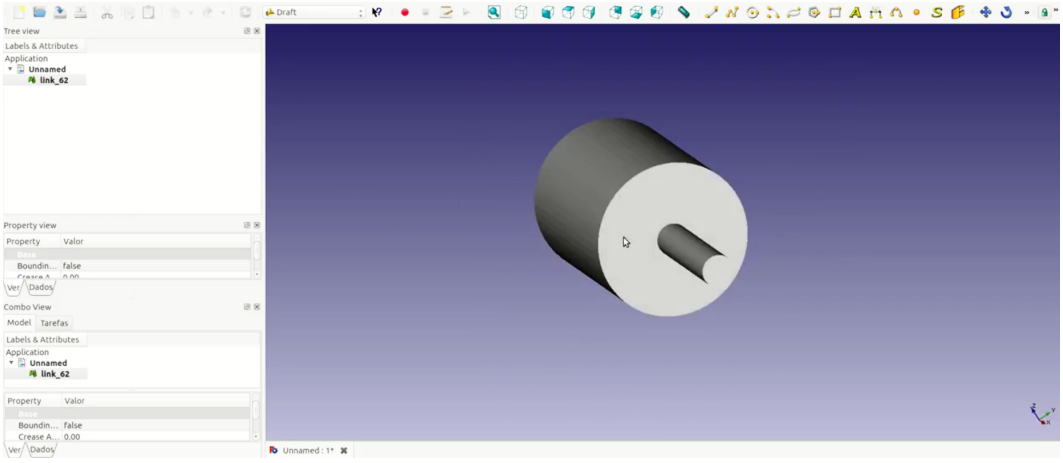


Figura 2.11: Utilização do FreeCAD para o modelo do *end-effector* do robô.

2.7 OpenCV

Open Source Computer Vision Library (OpenCV) é uma das mais populares bibliotecas de visão computacional, apresentada por Bradski [15] em 2000. Contém uma licença BSD e é gratis para aplicações académicas e comerciais. Pode ser programada, utilizando diversas linguagens de programaçao, C, C++, Python e Java, ajustando-se a diversas plataformas como o Windows, Linux, OSX, Android e iOS [16].

Para o ROS, esta biblioteca está disponível no `vision_opencv` que tem a seguinte estrutura de pastas:

- `cv_bridge`: esta *package* contém API que converte a mensagem de uma imagem em OpenCV para uma mensagem ROS, chamada `sensor_msgs/Image`, e vice versa. Resumindo, atua como uma ponte entre o OpenCV e o ROS. Sempre que quisermos enviar um imagem de um nodo para outro, torna-se necessário converter e processar a informação da imagem (figura 2.12).

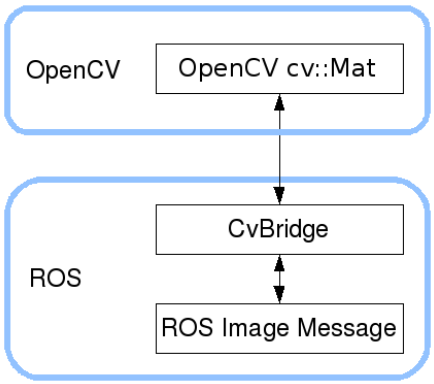


Figura 2.12: Converter imagens usando CvBridge.

- `image_geometry`: um dos primeiros processos a realizar, antes de trabalhar com câmaras, é a calibração. `image_geometry` contém uma biblioteca escrita em C++ e Python que ajuda a corrigir a geometria de uma imagem, utilizando os parâmetros de calibração. A *package* utiliza a mensagem específica do tópico `sensor_msgs/CameraInfo` para retificar uma imagem, usando os parâmetros de calibração.

A biblioteca `cv_bridge` funciona respeitando passos bem específicos. Os passos importantes para que se implemente esta solução resumem-se a [17]:

- `subscrever` o tópico que contém a informação que se pretende utilizar, por exemplo uma imagem;
- `converter` a mensagem ROS do tópico numa imagem OpenCV, usando o CvBridge;
- `realizar` as operações pretendidas na imagem utilizando ferramentas e bibliotecas;
- `converter` a imagem do OpenCV para publicar num tópico;

2.8 PCL

A *Point Cloud Library* (PCL), apresentada por Rusu e Cousins [18], é uma biblioteca *open source* para manipular imagens 2D/3D e processar nuvens de pontos. Tal como a biblioteca OpenCV, também tem uma licença BSD para uso académico e comercial livre, aplicado a sistemas operativos como Linux, Windows, Mac OS e Android/iOS.

Uma nuvem de pontos pode ser adquirida por meio do sensor Kinect, Asus Xtion Pro, Intel Real Sense e outros sensores habilitados a recolher informação desta natureza.

A PCL está totalmente integrada em ambiente ROS para a manipulação de dados de nuvens de pontos dos vários sensores. Existe uma interface do ROS para a biblioteca PCL denominada `perception_pcl`. Esta interface consiste em *packages* para processar os dados de nuvens de pontos de ROS para PCL e vice versa. A *package* `perception_pcl` contém a seguinte estrutura de pastas [17]:

- `pcl_conversions`: contém um conjunto de API para converter conteúdos PCL em mensagens ROS;
- `pcl_msgs`: inclui as mensagens trocadas entre PCL e ROS. As mensagens da PCL são:
  - ModelCoefficients;
  - PointIndices;
  - PolygonMesh;
  - Vertices;
- `pcl_ros`: integra ferramentas e nodos para fazer a ponte entre as mensagens ROS e o tipo de mensagens da PCL e vice versa.
- `pointcloud_to_laserscan`: comporta algoritmos que convertem a informação 3D de nuvens de pontos para laser scan 2D.

2.9 Linguagens de programação adotadas

Todo o desenvolvimento deste trabalho foi realizado em Ubuntu 14.04 LTS e em ambiente ROS. A programação desenvolvida foi em C++. Contudo, o algoritmo de inteligência de jogo de damas, foi desenvolvido em linguagem C. A escolha recaiu sobre a programação em C/C++ não só por ser a base de diversas linguagens de programação, mas também por permitir uma interação ao baixo nível do *hardware*.

Capítulo 3

Integração da API ROS-Industrial

Visto que nos projetos relacionados se verificou a existência de uma API disponível em código *open source* em ambiente ROS e com uma vasta aplicabilidade, procedeu-se à sua integração no robô industrial presente no LAR. Como as bibliotecas do ROS-Industrial se encontravam num estado avançado de desenvolvimento, a aprendizagem dos métodos e das ferramentas associadas ao ROS e a todas as suas ferramentas necessitam ser gradual.

O ROS-Industrial tem associado um conjunto de *metapackages* de fabricantes de robôs que contém não só a estrutura organizada de pastas e ficheiros de uma *metapackages*, mas também os modelos genéricos da geometria do robô. De entre as *packages* de fabricantes de robôs, selecionou-se a referente ao ROS Fanuc. Além deste trabalho utilizar um ambiente que serve como base de todo o *software* (o ROS), incluiu-se um segundo segundo ambiente que, de entre muitas funcionalidades, planeie trajetórias (o MoveIt!).

A estrutura deste capítulo inicia com a explicação do ROS seguindo-se o ROS-Industrial e o ROS Fanuc. Por fim, explicam-se as bibliotecas que compõe o MoveIt!.

3.1 ROS

ROS é uma plataforma de desenvolvimento que surgiu em 2007, fortemente impulsionada pela *Willow Garage*. Integra bibliotecas e ferramentas direcionadas a aplicações em robôs, simplificando tarefas complexas e robustas. Os *softwares* de diversas empresas estão, cada vez mais, a ser desenvolvidos em ambiente ROS, particularmente no ramo da robótica industrial [19].

O ROS torna-se uma excelente escolha para um projeto robótico, fornecendo diversas vantagens, como a capacidade de integrar recursos de navegação autónoma e recursos para planear movimentos, por exemplo, o MoveIt!. Oferece, de igual modo, ferramentas de *debugging*, visualização e simulação como o `rqt_gui`, RViz e Gazebo, respetivamente. Suporta interfaces para sensores e atuadores robóticos, podendo a estrutura ROS estabelecer comunicação entre vários nodos com diferentes linguagens de programação, como por exemplo C, C++ e Python [17].

O ROS está dividido em três grupos: o *file system level*, o *Computation Graph level* e o *Community level* (figura 3.1).

O primeiro nível, o *file system level*, define o funcionamento interno, a estrutura de armazenamento, a estrutura dos ficheiros e os requisitos mínimos ao seu funcionamento. É neste nível que são especificados detalhes das mensagens, serviços e pastas.

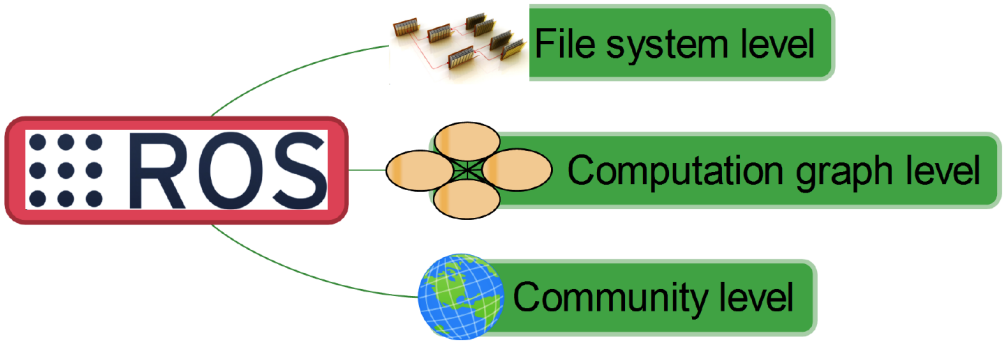


Figura 3.1: Estrutura geral do ROS.

Seguidamente, o *Computation Graph level*, descreve a comunicação entre os processos e o sistema. É neste nível que são descritas as definições e explicações das relações entre todos os constituintes do ROS. Por último, o *Community level*, representa a comunidade *online* que contém os algoritmos e o código que são destinados à partilha. Não será descrito o conteúdo deste tópico, uma vez que se refere a *sites online* que integram *packages* e *meta packages* [20].

File system level

As *packages* são o núcleo central do ROS e de cada *metapackage*. As *metapackages* são utilizadas para gerar um grupo de *packages*, criadas para processar e executar uma determinada tarefa. De forma a manter uma organização, tanto das pastas como do conteúdo da informação partilhada com a comunidade ROS, definiu-se uma estrutura típica para cada *package*, como está representada na figura 3.2.

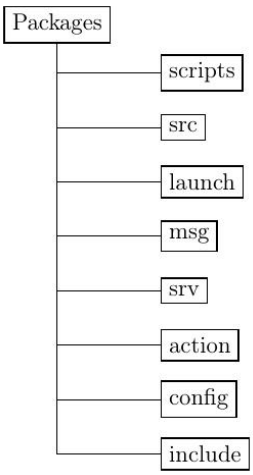


Figura 3.2: Estrutura e organização das pastas de cada *package*.

As pastas típicas que compõem uma *package* em ROS tem a seguinte estrutura [17]:

- todo o código desenvolvido em C++ está na pasta src, enquanto que o código desenvolvido Pyton está presente na pasta *scripts*;
- quando se pretende dar início a mais que um programa e/ou biblioteca, em simultâneo, utilizam-se ficheiros *launch*. Estes ficheiros estão na pasta *launch*;
- a pasta msg comporta a definição de todas as mensagens desenvolvidas pelo programador. Por sua vez, a pasta srv, possui todas as mensagens usadas no modelo cliente/servidor em ROS;
- a pasta *action* inclui os ficheiros para criar uma *action*. No presente trabalho não se recorreu a esta funcionalidade, uma vez que nunca houve a necessidade de criar esta pasta;
- os ficheiros de configuração de cada *package* estão presentes na pasta *config*;
- as *header files* utilizadas em cada nodo são guardadas na pasta *include*;

Computation graph level

O ROS cria uma rede que é responsável por manter toda a arquitetura interligada. Possibilita a cada nodo ROS o acesso à rede, podendo interagir, aceder e transmitir informação a outros nodos. Os conceitos do ROS *computational graph level* estão presentes na figura 3.3.

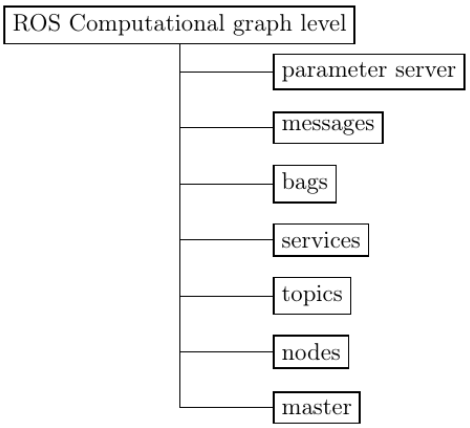


Figura 3.3: Estrutura do ROS Graph level.

Nos **nodos** ROS, onde todo o processo computacional é realizado, são construídos em roscpp(C++) ou rospy(Pyton). Cada robô poderá processar simultaneamente diferentes nodos e tarefas, estabelecendo comunicações entre si, utilizando tópicos, serviços e parâmetros. Com as metodologias disponíveis no ROS, torna-se possível implementar diferentes formas de comunicação entre nodos, possibilitando a troca de informação. Processos mais simples e mais leves computacionalmente são o grande objetivo dos nodos, evitando sobrecargas e elevados tempos computacionais. Se por alguma razão um nodo bloquear, todo o restante funcionalismo continua assegurado.

O **ROS *master*** tem um papel fundamental, fazendo a gestão de toda a arquitetura. Sem o desencadeamento desta funcionalidade, os nodos não conseguiam comunicar entre

si. ROS *master* possui a identidade de todos os nodos presentes no sistema, estabelecendo a troca de informação entre cada nodo ROS.

O ***parameter server*** permite ao utilizador alterar e configurar os parâmetros dos nodos enquanto estes estão em execução. Esta secção pertence ao ROS *master*.

Os nodos ROS comunicam entre si utilizando **mensagens**. As mensagens possuem uma estrutura de dados bem definida e conhecida entre cada nodo ROS. Recorre-se a uma estrutura padrão ou a uma estrutura criada manualmente para aplicações de troca de informação mais específica.

Cada mensagem é transportada utilizando **tópicos**. Quando um nodo ROS envia uma mensagem através de um tópico, considera-se que o nodo é um publicador. Por outro lado, quando um nodo ROS recebe uma mensagem proveniente de um tópico, define-se como sendo um nodo subscritor. O modelo publicador subscritor é uma forma de transmissão de dados unidirecional, isto é, é uma forma de transmissão de dados que apenas transmite informação para a rede sem *feedback*.

Bags é um formato que armazena todas as mensagens de tópicos e de serviços, dando a possibilidade de *play back* de um período de tempo definido. No presente trabalho não se deu utilidade a esta funcionalidade.

Em algumas aplicações é imprescindível realizar pedidos e aguardar por uma resposta, algo que o modelo publicador/subscritor não garante. Utilizando o modelo cliente/servidor estrutura-se um serviço que está dividido em duas partes, pedidos e respostas. O nodo cliente envia uma mensagem com um pedido para um nodo servidor. Enquanto o nodo servidor processa a mensagem e executa o **serviço**, o cliente aguarda pelo resultado. Após concluir, o servidor envia a resposta para o cliente. Neste modelo a comunicação estabelece-se, unicamente, entre dois nodos [20].

3.2 ROS Industrial

Até agora, discutiram-se as funcionalidades do ROS, aplicáveis a robôs didáticos e para investigação. No entanto, e face a toda a industrialização, pretende-se desenvolver uma aplicação ao nível industrial.

O projeto ROS-Industrial teve início em Janeiro 2012 com a grande colaboração das *Yaskawa Motoman Robotics*, *Willow Garage* e *Southwest Research Institute*. Com o ROS-Industrial é possível combinar as vantagens fornecidas pelo ROS às tecnologias e à segurança industrial a fim de explorar as grandes capacidades em processos de fabrico. Um *software* robusto e fiável para aplicações industriais, suportado por uma comunidade de investigadores e profissionais são os principais objetivos do ROS-Industrial [21].

A utilização do ROS-Industrial fornece inúmeras vantagens, destacando-se:

- **exploração de recursos:** as *packages* estão vinculadas à biblioteca ROS, possibilitando a sua utilização em diferentes robôs industriais. A plataforma ROS também é rica em ferramentas, tais como RViz e Gazebo para visualização e simulação;
- **aplicações *out-of-the-box***: a interface ROS permite a perceção avançada para trabalhar em aplicações de, por exemplo, *pick and place* de objetos complexos;
- **simples e económico:** dada a facilidade em calcular trajetórias entre posições e livre de colisões, o ROS-Industrial permite realizar aplicações robustas. Sendo este um *software open source*, possibilita o uso comercial sem qualquer tipo de restrições

ou mesmo licenças, fazendo com que a entidade proprietária do robô não dependa exclusivamente do *software* do fabricante;

O diagrama da figura 3.4 apresenta uma representação das *packages* que estão organizadas no topo do ROS.

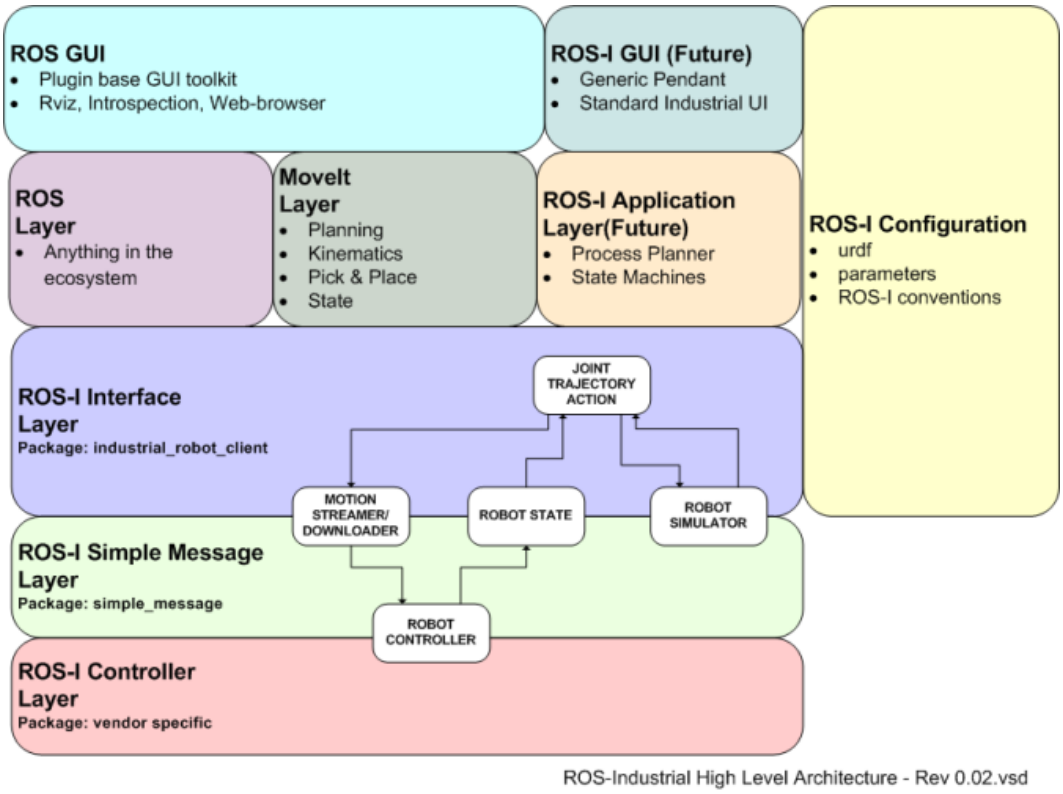


Figura 3.4: Diagrama de blocos da arquitetura de alto nível do ROS-Industrial [21].

De seguida, será feita uma breve descrição das *packages* actuais, presentes no ROS-Industrial [21].

- **ROS GUI** inclui todas as ferramentas de interface gráfica baseada em *plugins* ROS, tais como o RViz e Gazebo;
- **ROS Layer** é a camada de base onde são estabelecidas e executadas todas as comunicações;
- **MoveIt! Layer** oferece soluções para planear trajetórias, calcular cinemáticas inversas e diretas e soluções de *pick and place* na interface RViz;
- **ROS-I interface layer** consiste no cliente do robô. Estabelece a conexão com o controlador do manipulador, utilizando o protocolo *simple message*;
- **ROS-I simple message** é a camada de comunicação com o robô industrial. É um protocolo padrão que troca dados do cliente para o controlador e vice versa;

- **ROS-I controller** é uma unidade específica e reservada da informação do fabricante do robô;
- **ROS-I configuration** define não só os parâmetros de construção como o *design* e o modelo do robô em causa;

3.2.1 URDF para um robô industrial

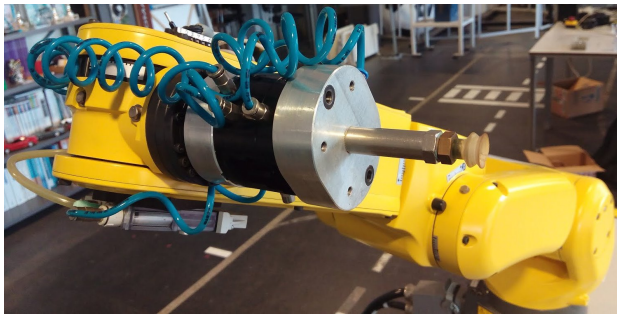
O *Unified Robot Description Format* (URDF) contém a informação e os formatos para representar o modelo virtual de um robô. Existem bastantes similaridades na criação de um ficheiro URDF para robôs didáticos e robôs industriais. As principais similaridades são ao nível da organização. As diferenças cingem-se a uma estrutura muito particular que deve ser respeitada durante uma modelação. A estrutura é a seguinte: [17]

- o algoritmo de cinemática inversa, IKFast, planeia trajetórias e salvaguarda a colisão entre os elos do manipulador bem como a colisão com objetos do meio ambiente. Por isso, cada modelo deverá conter uma descrição e dimensão o mais real possível de cada elo;
- cada elo tem um sistema de coordenadas próprio, onde o eixo x aponta na direção do comprimento, o eixo y na direção da largura e de forma a que o eixo z, resultante do produto externo entre os dois últimos, seja positivo para fora da folha. Estas referências são impostas quando o manipulador se encontra na *home position*;
- em ROS-Industrial, os modelos devem ser o mais detalhado possível, de forma a que a visualização no RViz seja detalhada. Por outro lado, o modelo utilizado para fazer a verificação de colisões deve ser mais leve e com menos detalhes de malha para que diminua o esforço computacional;
- cada junta do manipulador tem disponível 3 orientações (Roll, Pitch e Yaw), sendo que um dos valores não está limitado, dando a possibilidade de uma única rotação;
- o modelo total de um manipulador é escrito num único ficheiro macro. Neste estão disponíveis detalhes e especificações dos vários elos que o compõem. Foi neste ficheiro que se ajustou a posição do TCP do modelo do manipulador;
- em ROS-Industrial existem regras para a nomenclatura dos sistemas de coordenadas. O primeiro sistema de coordenadas define-se como `base_link`, que deve ser coincidente com a base do manipulador. O sistema de coordenadas do último elo denomina-se `tool0` (end-effector). É necessário garantir a coincidência do TCP do modelo com o TCP do manipulador real, em posição e orientação, de forma a que os cálculos realizados para o modelo sejam iguais para o robô real. Por defeito o *end-effector* vem desconfigurado.

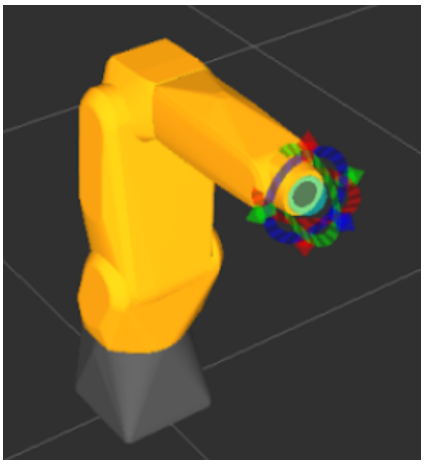
3.2.1.1 Implementação do modelo URDF *end-effector*

O robô é composto por 6 elos (figura 3.5a) sendo que no elo correspondente ao *end-effector* está acoplada uma ferramenta que faz a interação com o meio envolvente. Como esta ferramenta pode ser facilmente substituível, o modelo original do robô (figura 3.5b) não incorpora o modelo da ferramenta. Uma vez que se pretende uma elevada precisão

da extremidade (TCP) do *end-effector* para a aplicação final, decidiu-se não só atualizar o TCP virtual para corresponder à posição do TCP real, mas também incluir o modelo do *end-effector* completo.



(a) *End-effector* real.



(b) Modelo original do robô.

Figura 3.5: Comparação entre o modelo do *end-effector* do robô real e do modelo original do robô em ROS.

A representação inicial do último elo era constituído por um cilindro. O modelo virtual do *end-effector* foi construído em SolidWorks respeitando as medidas do mesmo (figura 3.6). Após a construção do modelo, procedeu-se à sua inclusão nos ficheiros URDF. Existem dois tipos de modelos, os modelos para colisão e os modelos para a visualização. Dado que o modelo CAD desenvolvido em SolidWorks é composto por 2500 pontos e 7500 arestas, torna-se pesado computacionalmente quando utilizado como modelo de colisão. Para tal, recorreu-se ao FreeCAD para criar um segundo modelo, análogo ao modelo CAD mas com menos detalhes e com uma malha mais grosseira.

Figura 3.6: Modelo do último elo do robô, correspondente ao *end-effector*.

Uma vez que a finalidade dos modelos para efeitos visuais não interferem com o cálculo, utilizou-se o modelo com a malha mais refinada.

3.2.2 Industrial Core

A *metapackage Industrial Core* [22] contém bibliotecas específicas para, por exemplo, comunicar com controladores industriais. Nesta *metapackage* estão presentes diversas *packages* que são responsáveis pelo seu funcionamento. Seguidamente, será apresentado um esquema da estrutura de *packages* que compõe a *metapackage* Industrial Core (figura 3.7).

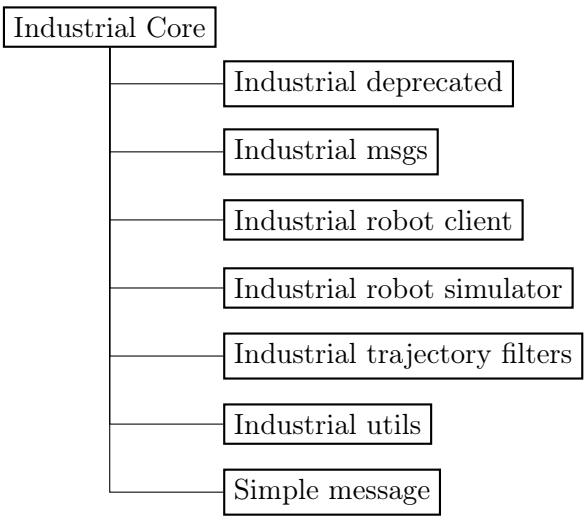


Figura 3.7: Estrutura de *packages* que compõe a *Metapackage industrial core*.

A *package* **industrial deprecated** possui toda a informação tanto dos autores como das dependências das pastas. **Industrial msgs** contém grande parte das mensagens específicas para aplicações industriais. Estas mensagens dividem-se em duas categorias, direcionadas para o modelo cliente/servidor e para o modelo publicador/subscritor.

A *package* **industrial robot client** integra bibliotecas de clientes para conetar com servidores alojados nos controladores industriais. Os nodos ROS contidos nos clientes são responsáveis por enviar dados de posição e trajetórias para o controlador de um robô industrial.

A *package* industrial robot client inclui os dois principais nodos ROS, o nodo `robot_state` e o nodo `joint_trajectory`. O primeiro responsabiliza-se por publicar o estado e a posição corrente do robô. Por outro lado, o segundo, subscreve o tópico de comando e envia a ordem para o controlador.

A *package* **industrial robot simulator** simula um controlador de um manipulador industrial. Este simulador apresenta apenas os requisitos mínimos presentes num controlador. A criação deste nodo torna possível, ao utilizador, desenvolver aplicações e movimentos sem a necessidade de equipamento físico. Este simulador simula e publica tópicos que podem ser acedidos pela ferramenta de visualização 3D, RViz. Desta forma, cria-se uma visualização realística da célula atual do robô. Esta interface de visualização não aceita conexões do protocolo *simple message*.

A *package* **industrial trajectory filters** contém métodos para filtrar as trajetórias de manipuladores. Estes filtros estão focados nas necessidades dos robôs e nas suas apli-

cações. Um exemplo de um filtro é o `uniform_sample_filter`. Este filtro recalcula um novo planeamento de uma trajetória com um tempo uniforme entre os pontos espalhados pelo trajeto.

A *package* **industrial utils** fornece uma biblioteca para detetar erros que possam gerar falhas e problemas durante o tempo de execução.

A *package* **simple message** define mensagens bem definidas para a conexão com controladores industriais. Como este protocolo de comunicação tem particular interesse, a sua explicação será detalhada adiante.

3.2.2.1 Simple message

O protocolo *simple message* [23] define uma estrutura de mensagem que é trocada entre o sistema cliente (o computador) e o servidor (o controlador do robô industrial). Para que a troca de informação entre as duas entidades seja fiável, cria-se uma mensagem que seja reconhecida por ambas as partes. A mensagem trocada entre o computador e o controlador é composta por 3 secções: o prefixo, o cabeçalho e o corpo. No prefixo é definido o tamanho da mensagem, enquanto que no cabeçalho define-se se aguarda por algum *feedback*, o tipo de comunicação e o género de mensagem. Neste último, refere-se o número do fabricante e das estandardizações. No corpo da mensagem segue a informação que se pretende trocar.

Um exemplo de uma mensagem que é trocada entre o cliente e o servidor está presente na tabela 3.1 e especifica a posição das juntas do robô. Nela, estão presentes todas as partes descritas anteriormente. Em cada um é definido o tamanho, o valor e o tipo.

Tabela 3.1: Estrutura da mensagem de juntas [23].

Member	**Type**	**Value**	**Size**
Message Type:	StandardMsgType::JOINT_POSITION	10	4 bytes
Communications Type:	CommType	ANY	4 bytes
Reply Type:	ReplyType	ANY	4 bytes
Data (Topic, Requests, & Response)			
sequence	shared_int	ANY	4 bytes
joints	shared_real[10]	ANY	40 bytes

3.3 ROS FANUC

ROS-Industrial *support packages* define uma organização que é transversal a todos os robôs industriais de diferentes fabricantes, facilitando a sua manutenção e atualização. No ROS-Industrial, existe um conjunto de *metapackage* de fabricantes de robôs. De todas as *metapackages* já desenvolvidas, selecionou-se a biblioteca Fanuc, focada no LR Mate 200iD. O esquema da figura 3.8 apresenta uma lista das *packages*, pastas e ficheiros que constituem a *Metapackage* ROS Fanuc original utilizada no trabalho.

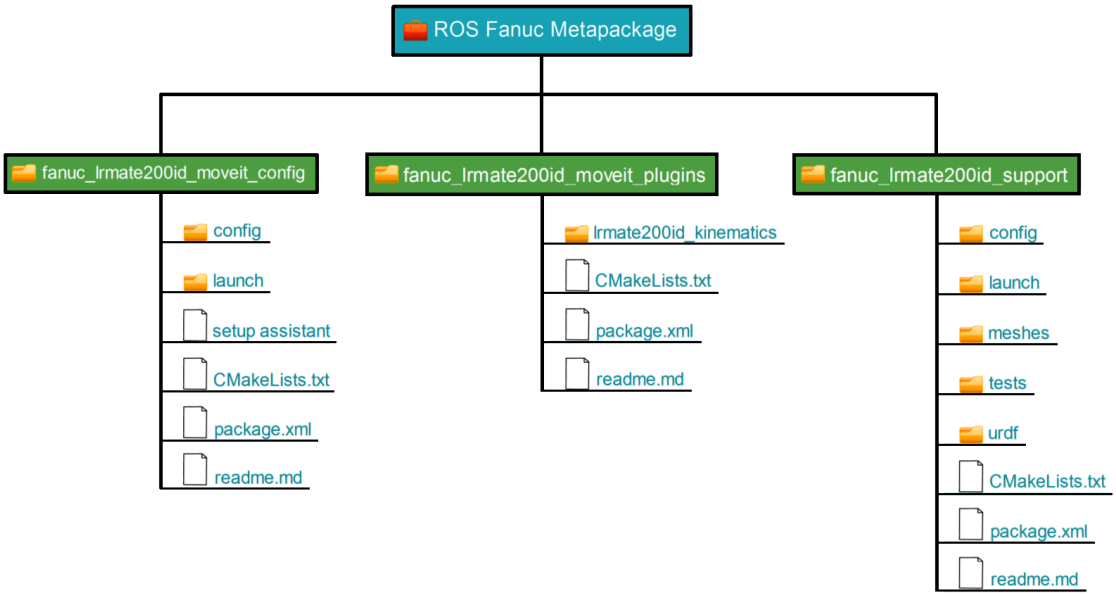


Figura 3.8: Estrutura original de organização das *packages* (a verde) que compõe a *metapackage* ROS Fanuc, focada no LR Mate 200iD [24].

As três *packages* que compõem a *metapackage* ROS Fanuc, `fanuc_lrmate200id_moveit_config`, `fanuc_lrmate200id_moveit_plugins` e `fanuc_lrmate200id_support` são as *packages* essenciais que contêm todas as pastas e ficheiros de configuração, cálculo de cinemáticas, modelos e disposição dos elos do robô.

A primeira pasta, da *package* `fanuc_lrmate200id_moveit_config`, integra ficheiros que atribuem os nomes aos elos, as juntas e o algoritmo de cinemática inversa utilizado. No fundo, contém toda a informação fornecida no assistente de configuração. Como será explicado na secção 3.4.7 deste capítulo, o assistente de configuração gera todos os arquivos de configuração e *scripts* de inicialização com base nas informações fornecidas pelo utilizador. A pasta *launch* inclui não só os ficheiros e configurações do Rviz mas também todos os ficheiros *launch* que fazem o arranque do cliente que conecta com o servidor do controlador.

A *package* `fanuc_lrmate200id_moveit_plugins` inclui, unicamente, o algoritmo *plugin* IKFast, responsável pelo cálculo da cinemática inversa.

Por último, a *package* `fanuc_lrmate200id_support` tem os seguintes ficheiros, distribuídos pelas seguintes pastas [17]:

- **config** - contém um ficheiro de configuração, `joint_names_lrmate200id.yaml`, que descreve o nome das juntas que compõe um robô;

- **launch** - contém ficheiros com as definições do manipulador industrial. Seguidamente, será feita uma descrição de cada um dos ficheiros:

  - `load_lrmate200id.launch` - este ficheiro carrega a descrição física do robô. De acordo com a complexidade do robô o número de ficheiros varia;

  - `test_lrmate200id.launch` - é responsável pela visualização do modelo URDF no RViz;

  - `robot_state_visualize_lrmate200id.launch` - lança e visualiza a posição corrente do robô. O estado atual do robô é visualizado através do RViz e do nodo `robot_state_publisher`. Uma das condições para que este nodo funcione é a dependência de um robô real ou simulado. Um dos argumentos a serem passados é o endereço IP do controlador industrial;

  - `robot_interface_streaming_lrmate200id.launch` - estabelece a comunicação bidirecional entre o servidor (controlador industrial) e o cliente (ROS_Industrial) e vice versa. Tal como o ficheiro anterior, este também requer um endereço IP, proveniente de um robô real ou simulado;

- **meshes** - contém todos os modelos em formato STL para a visualização e para colisões. Foi nesta pasta que se adicionou o modelo STL do *end-effector*;

- **testes** - inclui um ficheiro *launch* que é utilizado para testes;

Na figura 3.9 está presente a disposição dos nodos ROS e tópicos presentes na execução do *launch* `roslaunch fanuc_lrmate200id_moveit_config moveit_planning_execution.launch sim:=false robot_ip:=192.168.0.231`. Este é o principal ficheiro *launch* a ser executado uma vez que nele estão presentes todas as configurações do visualizador, o endereço IP, a construção do modelo no Rviz, o início do cliente e a conexão com o servidor. De seguida, será analisada a disposição da figura 3.9, com a explicação focada nos nodos.

Primeiramente, surgem três tópicos que são publicados pelo mesmo publicador `joint_states`. O tópico `feedback_states` possui uma mensagem que permite o *feedback* da posição corrente das juntas e a posição final. O tópico `robot_status` promove o estado corrente dos parâmetros do robô. Por último, o `joint_states` fornece a posição e a velocidade corrente das juntas do manipulador, quando este se encontra em movimento. Este último tópico é subscrito pelo `robot_state_publisher` que calcula todas as transformações geométricas e publica no `move group`. Estes três tópicos são atualizados com uma frequência entre 10-50 Hz, dependendo do *hardware* do controlador.

No campo do controlo de movimento surgem dois tópicos. O primeiro, intitula-se de `joint_path_command` e é o responsável por transportar a informação para executar a trajetória pré-calculada. Este é utilizado pelo `joint_trajectory_action` que gera trajetórias e publica comandos de movimento. Este último nodo publica em dois tópicos que são analisados pelos nodos `move_group` e pelo `motion_streaming_interface`. O `motion_streaming_interface` constrói uma mensagem para enviar para o controlador que contém conjuntos de trajetórias.

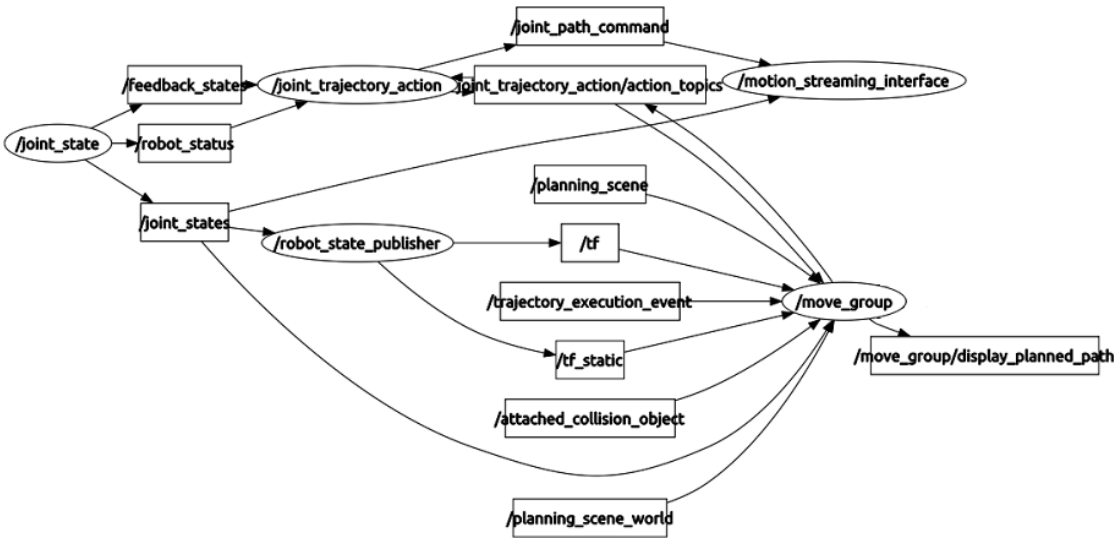


Figura 3.9: Disposição dos nodos ROS (a oval) e tópicos (em retângulo), com a conexão ao servidor já estabelecida.

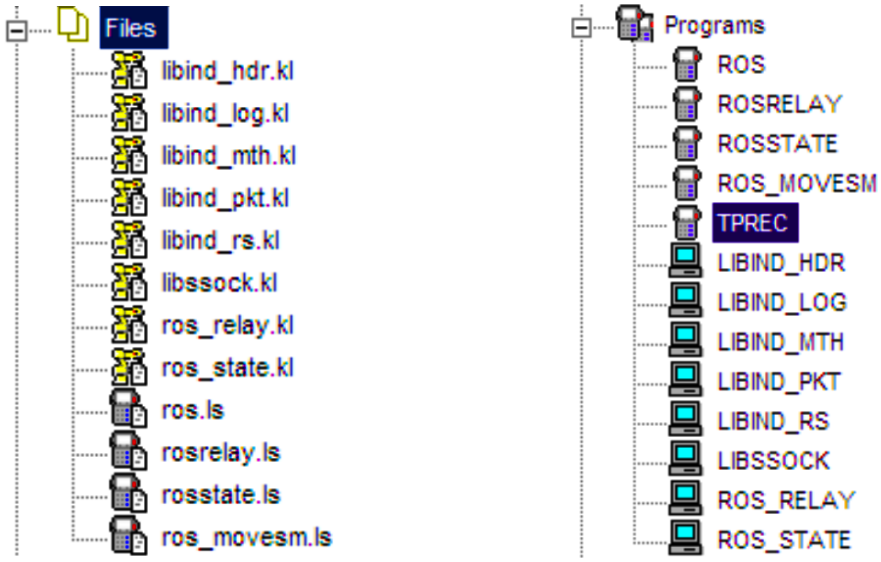
3.3.1 Instalação do servidor

De forma a instalar o servidor do ROS-Industrial no controlador Fanuc do LAR, utilizou-se como auxílio os tutoriais disponíveis no ROS-Industrial [25]. A informação trocada entre o servidor e o cliente é em espaço de juntas, ou seja, é trocada informação de cada uma das seis juntas do robô. Do mesmo modo, os movimentos executados pelo robô também são no espaço de juntas. Antes de iniciar qualquer tipo de alteração no controlador, sugere-se criar uma cópia de segurança bem como verificar se o controlador suporta ficheiros KAREL e *User Socket Messaging* (USM). O procedimento está dividido em 3 grupos: instalação, configuração e execução. De forma resumida, descreve-se o procedimento implementado.

Inicialmente, criou-se uma célula de trabalho nova e vazia no Roboguide, *software* fornecido pela Fanuc. Nesta célula importam-se (figura 3.10a) e compilam-se (figura 3.10b) todos os programas fornecidos na pasta `fanuc_driver` [26].

Existem dois métodos para a transferência dos ficheiros compilados para o controlador. O primeiro método consiste em utilizar a conexão FTP. Como a implementação do servidor foi uma das primeiras tarefas realizadas no trabalho, não se tinha um conhecimento tão profundo da transferência de programas para o controlador via FTP. O segundo método resume-se na transferência de ficheiros para o controlador via USB. Este método intitula-se *binary install* e foi o adotado na transferência de dados para o controlador.

De seguida, procedeu-se à configuração do servidor e dos ficheiros transferidos. Na configuração do servidor são precisos dois ficheiros FTP e de dois ficheiros USM. No trabalho de Cancela [2], já tinham sido criados os quatro ficheiros explicados anteriormente. Como o protocolo de comunicação era igual ao implementado no trabalho de Cancela, mantiveram-se em execução os dois ficheiros FTP, que servem de base para a conexão TCP/IP. Por outro lado, interrompeu-se a execução dos ficheiros USM existentes, sendo



(a) Ficheiros importados.

(b) Ficheiros compilados.

Figura 3.10: Ficheiros da pasta fanuc driver, importados e compilados, no Roboguide.

necessário criar dois novos e específicos para o servidor ROS. A figura 3.11 apresenta seis ficheiros. Os quatro primeiros pertenciam ao servidor do trabalho de Cancela, enquanto que os dois últimos pertencem ao servidor deste trabalho. Na coluna state da mesma figura, verifica-se que apenas estão em execução os primeiro, segundo, quinto e sexto.

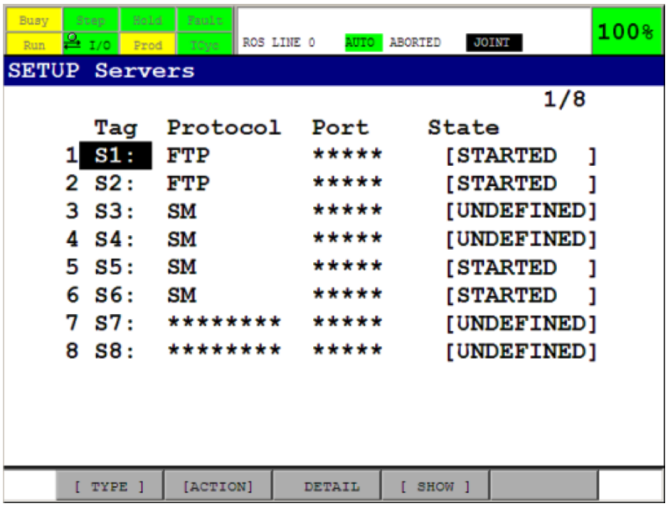


Figura 3.11: Janela de configuração, presente na consola, que contém os ficheiros da comunicação.

Configuraram-se os valores padrão dos ficheiros transferidos especificando, por exemplo, a frequência de atualização dos dados, a porta de comunicação, o número do ficheiro do servidor, entre outras. As configurações foram realizadas nos dois principais ficheiros transferidos para o controlador que servem de base do servidor, o `ros_relay` (figura 3.12)

e o `ros_state` (figura 3.13). Quando o utilizador termina a configuração dos ficheiros, é necessário alterar-se o estado da linha *checked*, de *false* para *true*.

Name	Type	Default	Unit	Description
checked	BOOLEAN	False	-	Configuration has been completed by user
f_msm_rdy	INTEGER	1	-	movesm i'face: 'ready/ack' signal flag
f_msm_drdy	INTEGER	2	-	movesm i'face: 'data ready' signal flag
loop_hz	INTEGER	40	Hz	Main loop update rate
move_cnt	INTEGER	50	%	CNT to set with each joint motion instruction
move_speed	INTEGER	20	%	Joint speed to set for all trajectory points
pr_move	INTEGER	1	-	movesm i'face: position register for next trajectory point
r_move_spd	INTEGER	1	-	movesm i'face: integer register for motion speed
r_move_cnt	INTEGER	2	-	movesm i'face: integer register for CNT value
s_tcp_nr	INTEGER	11000	-	TCP port to listen on
s_tag_nr	INTEGER	4	-	Index of the Server Tag to use
um_clear	BOOLEAN	True	-	Clear user menu on start

Figura 3.12: Configuração do ficheiro ros_relay transferido de modo a especificar todos os parâmetros utilizados pelo servidor.

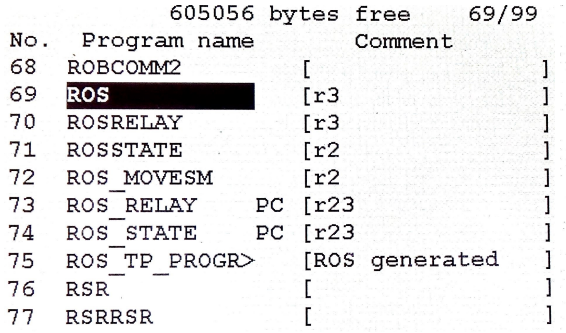
Name	Type	Default	Unit	Description
checked	BOOLEAN	False	-	Configuration has been completed by user
loop_hz	INTEGER	40	Hz	Main loop update rate
sloop_div	INTEGER	10	-	Divider for robot_status reporter loop
s_tcp_nr	INTEGER	11002	-	TCP port to listen on
s_tag_nr	INTEGER	3	-	Index of the Server Tag to use
um_clear	BOOLEAN	True	-	Clear user menu on start

Figura 3.13: Configuração do ficheiro ros_state transferido de modo a especificar todos os parâmetros utilizados pelo servidor.

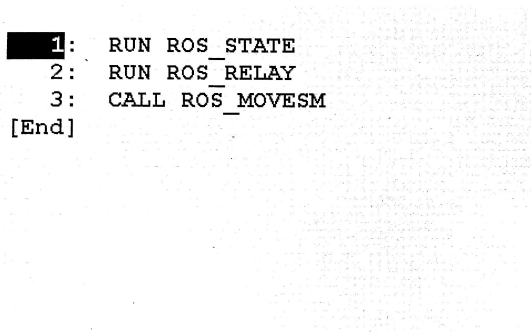
Por último, verifica-se a operacionalidade de toda a implementação. Executa-se o programa ROS TPE em ciclo automático e no computador procede-se à execução de um *roslaunch* específico à operação pretendida. O ficheiro ROS transferido é o ficheiro global que contém todos os ficheiros base.

3.3.1.1 Arranque do servidor

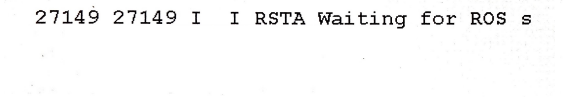
O arranque do servidor ROS é realizado com base num programa que foi transferido para o controlador. O programa intitula-se ROS (figura 3.14a) e é constituido por três linhas de código (figura 3.14b). As duas primeiras têm o conteúdo explicado na secção anterior, enquanto que a última fica a aguardar pela conexão de um cliente. Após se proceder à execução do programa, o servidor fica ativo e a aguardar pela conexão de um cliente (figura 3.14c). Quando a aplicação cliente é executada, verifica-se a conexão entre ambos (retângulo amarelo) e aguarda-se pela receção de mensagens (figura 3.14d).



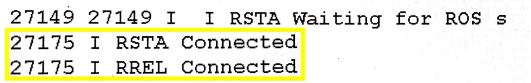
(a) Lista de programas no controlador.



(b) Conteúdo do programa ROS.



(c) Arranque do servidor.



(d) Conexão estabelecida.

Figura 3.14: Demonstração do arranque do servidor ROS. As imagens pertencem ao ambiente da consola.

3.3.2 Pós-processador Fanuc

O pós-processador Fanuc permite que o programador desenvolva programas LS/TP recorrendo à programação C++. Esta forma inovadora permite que sejam criados programas sem recorrer à consola nem a movimentos prévios do robô. Este pós-processador está totalmente integrado em ROS e é considerado como uma extensão a todo o pacote original. O projeto que deu origem ao pós-processador Fanuc foi desenvolvido pelo *Institut Maupertuis* [27].

A tabela 3.2 contém todas as funções de funcionamento interno e de definições para um programa TP. Linguagem TP é um tipo de programação definida em consola. Exemplo de funções são o envio de programas do computador para o controlador, definir um tamanho de um programa TP ou mesmo atribuir um comentário sobre o programa TP.

Por outro lado, a tabela 3.3 contém todas as funções que compõem um programa TP. Exemplo das funções são criar um ponto, controlar o estado de um I/O ou criar um movimento em relação a um sistema de coordenadas específico.

Tabela 3.2: Funções de funcionamento interno e de definições para um programa TP.

Função	Significado
generateProgram	Gera um programa TP
clearProgram	Elimina um programa TP presente no controlador
uploadToFtp	Faz o *upload* do programa TP para o controlador
setProgramName	Define o nome do programa TP
setProgramComment	Gera um comentário sobre o programa TP
setMemorySize	Atribui um tamanho para o programa TP
setPermissions	Define as permissões de um programa TP
useLineNumbers	Define o número de linhas de um programa TP

Tabela 3.3: Funções disponíveis para criar o conteúdo de um programa TP.

Função	Significado
appendPoseFine	Cria uma posição com aproximação fina (movimento de juntas ou linear)
appendPoseCNT	Cria uma posição com aproximação ajustável (movimento de juntas ou linear)
appendComment	Insere um comentário
appendEmptyLine	Adiciona uma linha vazia
appendDigitalOutput	Altera o estado de uma saída digital
appendSetRegister	Altera o estado de um registo
appendWait	Atribui um tempo de espera com duração ou consoante uma entrada digital
appendUFrame	Atribui um movimento em relação a um sistema de coordenadas
appendUTool	Atribui o sistema de coordenadas do *end-effector*
appendGroupOutput	Altera o estado de uma saída de grupo
appendRun	Arranca com outro programa TP presente no controlador
appendLabel	Cria uma *label* para referência do *jump*
appendJumpLabel	Cria um salto para a linha de comando do *label*

A figura 3.15 enquadra o pós-processador. O utilizador desenvolve um programa em programação $C++$ diretamente no computador. Concluído o desenvolvimento de um programa, o utilizador apenas converte o programa, de linguagem $C++$ para linguagem TP, e envia o programa para o controlador. A transferência do ficheiro é realizada por FTP.

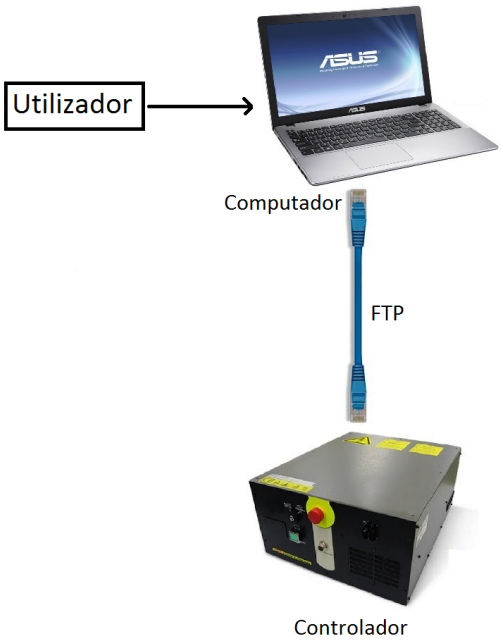
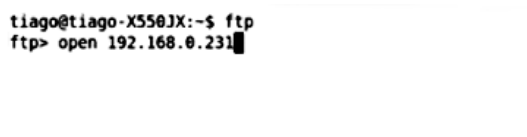


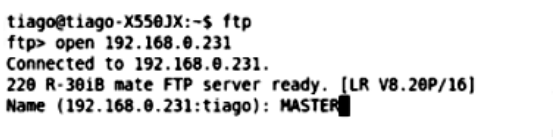
Figura 3.15: Esquema representativo do *hardware* necessário para utilizar o pós-processador.

A transferência dos ficheiros via FTP revelou-se um processo difícil de implementar. Para se transferirem ficheiros entre o computador e o controlador, eram exigidas credenciais de acesso (nome e *password*) por parte do controlador. Como o pós-processador original não dispunha desta opção, era impossível o envio de um programa TP. Após algumas trocas de emails com os autores, a solução passou por, antes de enviar o programa TP para o controlador, o pós-processador solicitar as credenciais.

Para garantir que a conetividade entre as duas entidades seja estabelecida, sugere-se o acesso via terminal. A verificação via terminal inicia com a abertura da conexão entre o cliente e o servidor via FTP (figura 3.16a). De seguida, são solicitados os dois parâmetros para as credenciais do servidor, o nome (figura 3.16b) e a *password* (figura 3.16c). Por fim, e para verificar se a conetividade foi estabelecida com sucesso, sugere-se o *download* de um programa TP presente no controlador (retângulo vermelho) e o *upload* de um programa TP gerado no computador para o controlador (retângulo amarelo) (figura 3.16d).

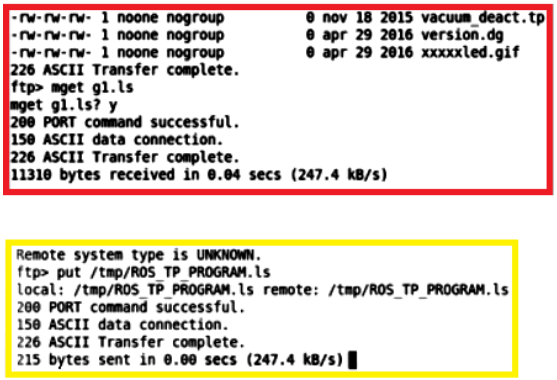


(a) Acesso ao servidor.



(b) Definir o nome da credencial.

(c) Definir a *password* da credencial.



(d) *Download* e *upload* de um programa.

Figura 3.16: Verificação da conexão entre o cliente e o servidor por FTP via terminal.

3.3.2.1 Criar e enviar um programa TP

A título de exemplo, criou-se um programa (figura 3.17) em linguagem de programação C++. As três primeiras linhas do programa dizem respeito às configurações típicas de um nodo em ROS. Após o espaço em branco começa o código para gerar um programa TP.

O programa inicia com as configurações necessárias para um programa TP. Neste caso atribuiu-se um nome ao programa TP, um comentário para descrever o programa e o número de linhas. Por defeito um programa é gerado com permissão para escrever e ler. Após serem concluídas todas as configurações, constrói-se o corpo do programa.

O corpo do programa, no caso do exemplo, inicia com um comentário. Posteriormente, altera-se o estado de registo, espera-se meio segundo e volta-se a alterar o estado do registo. Após se esperar mais nove segundos, adiciona-se uma linha em branco e cria-se um ponto no espaço para o robô se movimentar. Por fim, é executado um programa TP existente no controlador.

Terminada a programação do pós-processador, compila-se o programa gerado e, num novo terminal, executa-se o nodo ROS do programa criado. É gerado um ficheiro que contém o programa TP para ser enviado para o controlador (figura 3.18). O conteúdo do programa TP gerado tem a mesma informação do corpo do programa em C++.

A última etapa é o *upload* do programa do computador para o controlador via FTP. Após o programa ser transferido para o controlador (figura 3.19), este precisa de ser executado com recurso à consola.

```
int main(int argc, char **argv)
{
  std::string package = "fanuc_post_processor_application";
  ros::init(argc, argv, package);
  ros::NodeHandle node;

  FanucPostProcessor fanuc_pp;
  fanuc_pp.setProgramName("ros_tp_example_program");
  fanuc_pp.setProgramComment("ROS generated");
  fanuc_pp.useLineNumbers(false);
  fanuc_pp.appendComment("This is a ROS generated TP program");
  fanuc_pp.appendSetRegister(7, false);
  fanuc_pp.appendWait(0.5);
  fanuc_pp.appendSetRegister(7, true);
  fanuc_pp.appendWait((unsigned) 9);
  fanuc_pp.appendEmptyLine();
  fanuc_pp.appendPoseCNT(FanucPostProcessor::JOINT,
Eigen::Isometry3d::Identity(), 2, 20, FanucPostProcessor::PERCENTAGE, 100);
  fanuc_pp.appendRun("MY_OTHER_TP_PROGRAM");

  std::string program;
  fanuc_pp.generateProgram(program);
  ROS_WARN_STREAM("This is the generated program:\n\n" << program);

  if (!fanuc_pp.uploadToFtp("192.168.0.231"))
    return -1;
  return 0;
}
```

Figura 3.17: Programação C++ do pós-processador.

ROS_TP_EXAMPLE_PROGRAM.ls ×

```
/PROG ROS_TP_EXAMPLE_PROGRAM
/ATTR
COMMENT = "ROS generated";
PROTECT = READ_WRITE;
DEFAULT_GROUP = 1,*,*,*,*;
/MN
:  !This is a ROS generated TP program;
:  R[7]=0;
:  WAIT  0.5(sec);
:  R[7]=1;
:  WAIT  9(sec);
:  ;
:J P[2] 20% CNT100 ;
:  RUN MY_OTHER_TP_PROGRAM;
/POS
P[2]{
   GP1:
    UF : 0, UT : 1,         CONFIG : 'N U T, 0, 0, 0',
    X = 0.000000 mm, Y = 0.000000 mm, Z = 0.000000 mm,
    W = -0.000000 deg, P = 0.000000 deg, R = -0.000000 deg
};
/END
```

Figura 3.18: Linguagem TP, gerada pelo pós-processador, presente no computador.

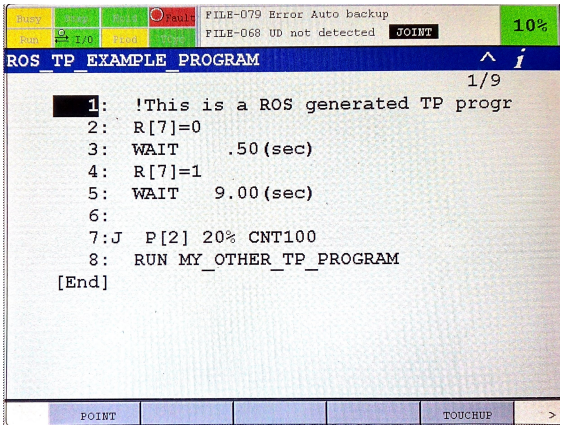


Figura 3.19: Presença do programa gerado pelo pós-processador no controlador.

3.3.3 Interface I/O

A interface I/O de um robô, especialmente em ambiente industrial, é uma excelente opção para controlar e comunicar com as unidades periféricas. No trabalho pretendia-se controlar o estado de uma eletroválvula que permitisse controlar o ar comprimido da ventosa do *end-effector*.

Após várias pesquisas e trocas de emails com o criador da *Metapackage* do ROS Fanuc, *G.A. vd. Hoorn*, concluiu-se que não existe, em ROS-Industrial, nenhuma interface de acesso aos I/O. De forma a solucionar o problema, estudou-se a possibilidade de implementar uma rede de campo baseada em ModBUS/TCP. ModBUS/TCP é uma solução transversal a todos os sistemas de automação, simples de implementar e com grande aplicação em ambiente industrial. No entanto, verificou-se que o controlador do robô, utilizado neste trabalho não estava atualizado e, portanto, não dispunha da opção ModBUS/TCP.

A solução adotada para movimentar uma peça de jogo foi deixar sempre atuada a saída digital correspondente ao acionamento da eletroválvula. Como a ventosa está sempre atuada, quando a ventosa fica em contacto com uma peça, esta fica agarrada ao *end-effector*. Para que o robô deixe uma peça numa posição, obstrui-se a tubagem de ar comprimido que alimenta o robô. Desta forma, torna-se exequível a movimentação de qualquer peça durante um jogo.

3.4 MoveIt!

O MoveIt! é um conjunto de *packages* e ferramentas para levar a cabo a manipulação dinâmica em ROS. A biblioteca MoveIt! contém o estado da arte da perceção 3D, o planeamento de trajetórias dinâmicas, a manipulação, as cinemáticas, a verificação de colisão, o controlo e a navegação. Além da interface utilizando API de programação, o MoveIt! fornece boas interfaces de interação entre o utilizador comum e o robô, já em utilização em mais de 65 robôs. Um exemplo disso é a interface em RViz que permite visualizar o planeamento dinâmico dos robôs [28].

Os utilizadores têm acesso a esta biblioteca por dois meios. O primeiro é utilizando

a interface gráfica presente no RViz (visualisador ROS). O segundo é recorrendo à programação, interagindo diretamente com os nodos principais. No trabalho recorreu-se aos dois meios disponíveis, sendo o primeiro usado numa fase mais inicial e o segundo para desenvolver a aplicação. Foi utilizada a programação $C++$ para interagir com o MoveIt!, focada nas classes *Move Group* [29] e *Planning Scene Interface* [30].

Esta secção aborda a interface gráfica e os constituintes do MoveIt!.

3.4.1 Interface RViz para MoveIt!

A interface RViz para MoveIt! (figura 3.20) é composta por duas grandes secções, o planeamento e o *plugin*. Esta interface inclui também a posição de cada junta do robô representada no modelo do robô.

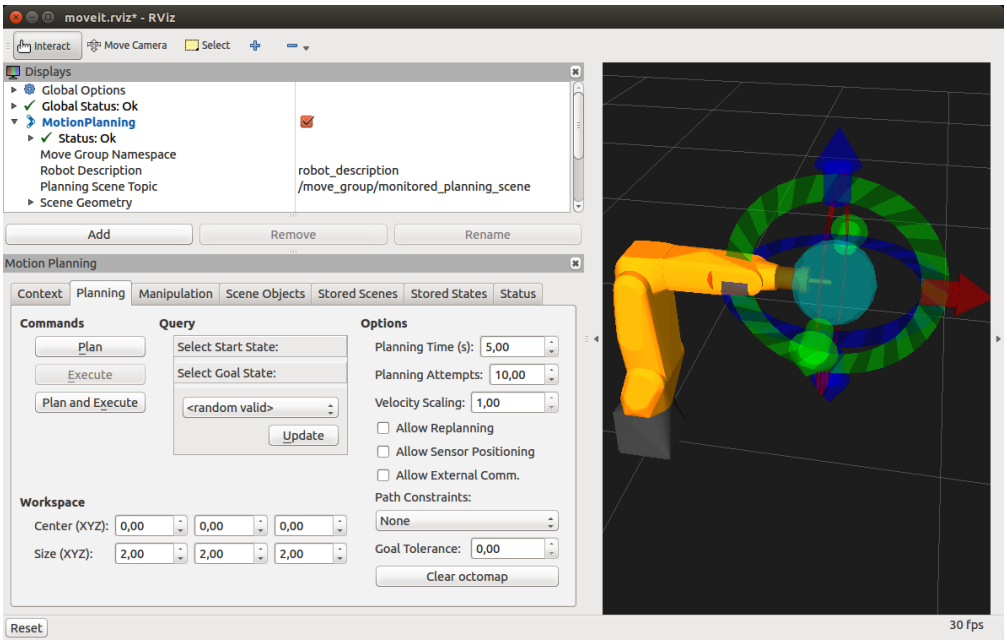


Figura 3.20: Interface RViz para Fanuc LR Mate 200iD.

De entre os separadores disponíveis, o separador de planeamento (figura 3.21) foi o utilizada no trabalho, uma vez que é o único que planeia trajetórias. Os restantes separadores permitem ações como manipular, adicionar e interagir com objetos na cena envolvente ao robô. No separador planeamento define-se o início, o fim e planeia-se o percurso que o braço robótico terá de percorrer. Por defeito, quando o visualizador inicia, as posições final e inicial são coincidentes. A orientação e posição do robô podem ser alteradas com a interação da marca interativa, coincidente com a extremidade do *end-effector*. De notar que esta interface previne situações de colisão com os próprios elos ou em posições onde o manipulador não atinja a posição pretendida.

O *plugin* centra-se na visualização. Na figura 3.22 estão presentes três regiões mais relevantes. A primeira define a presença do robô no visualizador, enquanto que a segunda possibilita a visualização do volume de trabalho do robô, as posições de início e fim e as cores para situações específicas. Por último, a terceira região subscreve o tópico do movimento da trajetória para se poder visualizar o percurso gradual.

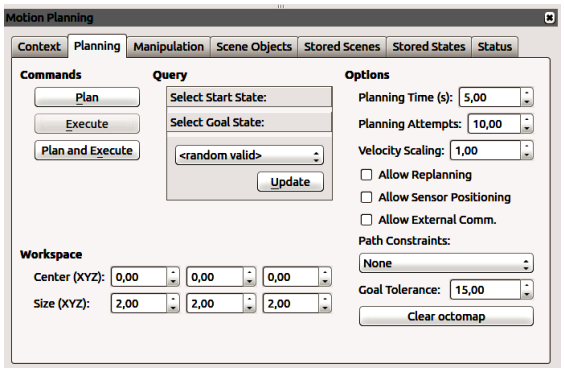


Figura 3.21: Separador para planear trajetórias do MoveIt!.

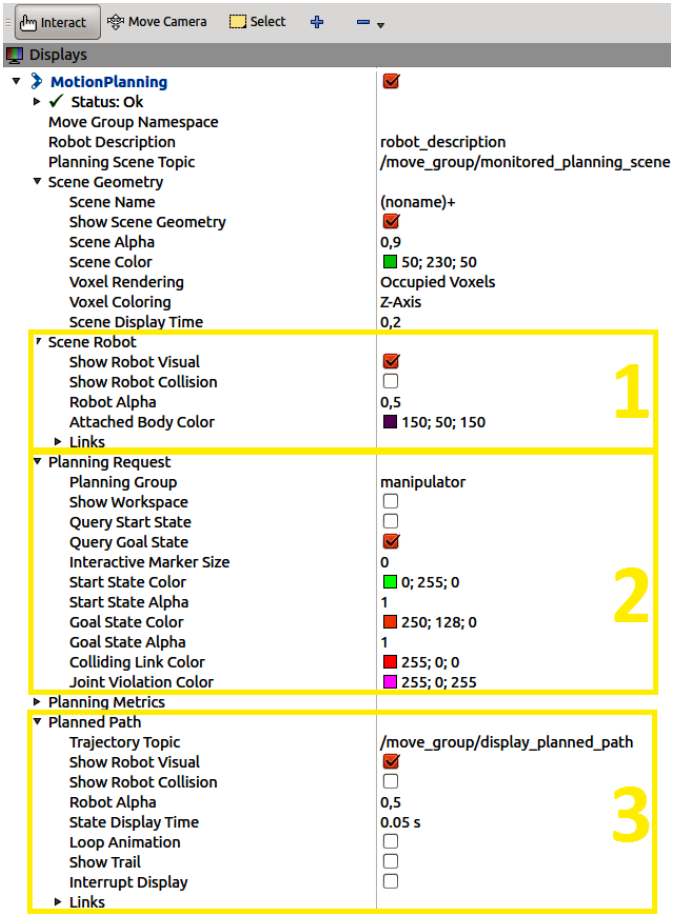


Figura 3.22: Definições do *plugin MotionPlanning*.

3.4.2 Arquitetura do MoveIt!

Na figura 3.23 está ilustrada a arquitetura de alto nível do MoveIt!. Esta arquitetura permite a comunicação com o servidor do robô, extraindo informações do seu estado (posições das juntas, etc), com sensores externos. É feita uma pequena descrição da arquitetura e dos conceitos base do MoveIt!.

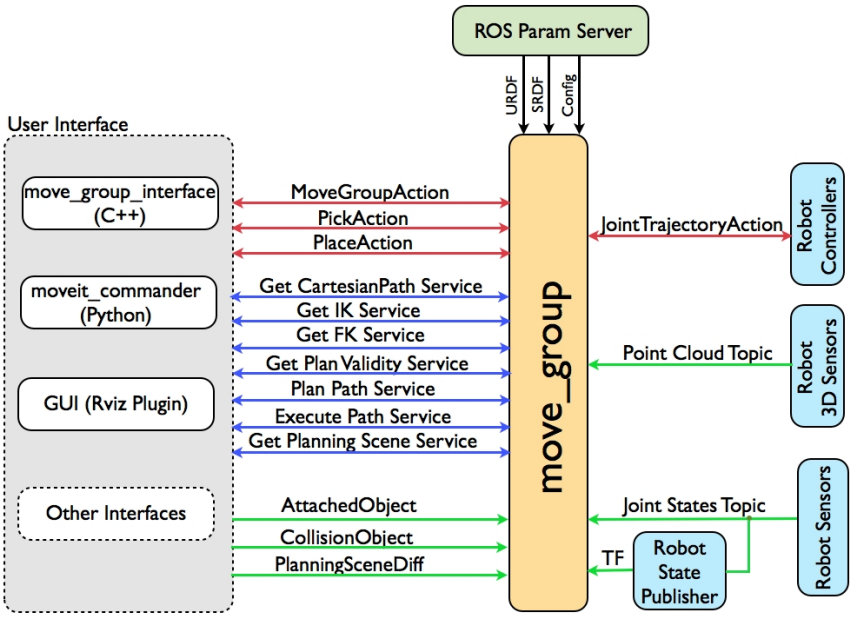


Figura 3.23: Diagrama da arquitetura do MoveIt! [28].

O coração do MoveIt! é o `move_group` que integra todos os componentes de informação. O ROS *Param Server* fornece ao `move_group` um conjunto de informações que serão enumeradas a seguir:

- **URDF**: o `move_group` analisa o parâmetro `robot_description` do ROS *Param Server* para examinar a informação do URDF do robô.
- **SRDF**: o `move_group` analisa o parâmetro `robot_description_semantic` no ROS *Param Server* para obter o *Semantic Robot Description Format* (SRDF) do robô. O SRDF contém a informação e as relações entre os modelos URDF.
- **MoveIt! configuration**: o `move_group` analisa o ROS *Param Server*, recolhendo informação dos limites teóricos das juntas e cinemáticas. Os ficheiros para estes componentes são automaticamente gerados pelo assistente de configuração do MoveIt!.

Quando o MoveIt! adquire todas as informações e configurações do robô, fica apto para ser comandado através das suas interfaces. `Move_group` pode ser comandado através de programação C++ ou Python. Desencadeia ações, como por exemplo *pick and place*,

cinemática direta e inversa. Este nodo não executa qualquer tipo de algoritmos de planeamento de movimento, apenas interliga todas as funcionalidades.

Os dados das transformações são processados pela biblioteca *Transform Frames* (TF) do ROS, a qual permite que um nodo tenha informação global do posicionamento do robô. Por exemplo, o ROS publica e disponibiliza as transformações ocorridas desde a base do manipulador até ao sistema de coordenadas do *end-effector*. Posteriormente, o `move_group` apenas recolhe a informação da transformação para a utilizar.

3.4.3 Planeamento de trajetórias

Planear trajetórias engloba o conjunto de estudos e métodos que permitem movimentar o manipulador. Os movimentos podem ser simples, como mover o *end-effector* entre duas posições em repouso no espaço de trabalho, durante um determinado período de tempo, ou complexos como planear trajetórias variáveis. A geometria geral do robô (URDF) e a descrição geral do ambiente são também tidas em conta.

O MoveIt! utiliza a interface *Open Motion Planning Library* (OMPL) para planear todas as trajetórias. OMPL [31] é uma biblioteca que contém o estado da arte de diversos algoritmos para planear trajetórias, utilizada em ambiente ROS. Dada a vasta aplicabilidade desta biblioteca, o cálculo de trajetórias não se encontrava especificada para problemas de robótica. Deste modo, foi adotado o algoritmo ***`RRTConnect`***, selecionado com base em resultados [32]. Esta configuração define uma infinidade de trajetos aleatórios e avalia-os. Após os avaliar, escolhe o caminho ótimo (em espaço das juntas) para movimentar gradualmente o robô entre duas posições.

3.4.4 Meio envolvente

O *planning scene monitor* (figura 3.24) é uma sub-parte do `move_group` representando o meio envolvente em torno do robô bem como o seu posicionamento. Recolhe informação das configurações das juntas do robô, do sensor (usualmente nuvens de pontos) e da geometria do ambiente envolvente, criada pelo utilizador.

O *planning scene monitor* é constituído por três subgrupos [28]:

- ***World geometry monitor*** constrói a representação tridimensional do meio ambiente em torno do robô. Utiliza o *octomap* para registar a informação adquirida. Regista também informação adicional introduzida pelo utilizador;
- ***Scene monitor*** acede ao tópico da informação das configurações das juntas do robô e cria uma representação mais rigorosa da posição do robô;
- ***State monitor*** pertence ao *World geometry monitor* centrando-se na informação de distâncias e nuvens de pontos;

No presente trabalho, utilizou-se a informação da posição das juntas do robô, fornecida pelo servidor do controlador e criaram-se os modelos dos objetos em torno do robô, como por exemplo a bancada de trabalho e o sensor.

3.4.5 Cinemáticas

Por defeito, o cálculo da cinemática inversa é auxiliado pela *package* ROS, *Kinematics and Dynamics Library* (KDL) [33]. Esta biblioteca é preferencialmente direcionada para

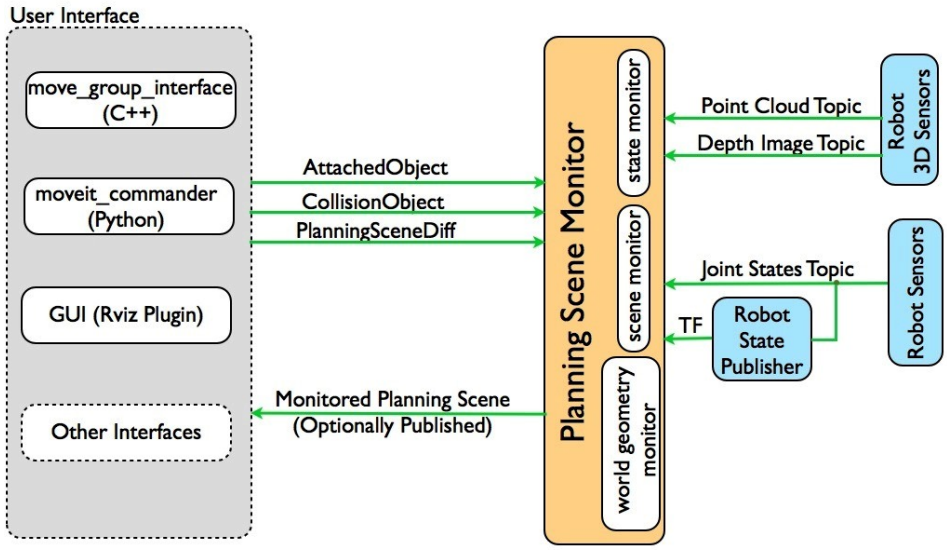


Figura 3.24: Diagrama do *planning scene monitor* que representa o meio envolvente em torno do robô.

manipuladores com 6 ou mais graus de liberdade. No caso de manipuladores com graus de liberdade menor ou igual a 6, pode ser adotada uma solução analítica mais rápida comparativamente ao KDL, o IKFast [34]. Dadas as suas vantagens, no presente trabalho adotou-se pela utilização do IKFast para o cálculo de cinemáticas inversas.

No caso das cinemáticas diretas e dos jacobianos, não é necessário o recurso a uma solução analitica, uma vez que estes já estão integradas na classe RoboState [35], `setFromIK` e `getJacobian`.

Para uma posição no espaço foram determinadas as cinemáticas diretas, inversas e o jacobiano (figura 3.25). Na cinemática direta, tal como na cinemática inversa, as posições estão em metros e os ângulos em radianos. Como se esperava, o jacobiano é quadrado, dado que o número de variáveis no espaço cartesiano é igual ao número de variáveis no espaço junta.

```
[ INFO] [1471365075.587087381]: Loading robot model 'fanuc_lrmate200id'...
[ INFO] [1471365075.827779940]: Loading robot model 'fanuc_lrmate200id'...
[ INFO] [1471365075.972942956]: Model frame: /base_link
 ***************Forward Kinematics***************
Translation:
-0.128373
-0.0677034
0.272965
Rotation:
  -0.775425 -0.0712848  -0.627403
  0.433043   0.663101   -0.61055
  0.459555  -0.745128  -0.483315

  ***************Inverse Kinematics***************
Joint joint_1: 1.885260
Joint joint_2: -1.043029
Joint joint_3: 3.732236
Joint joint_4: 0.726385
Joint joint_5: 2.001711
Joint joint_6: -2.099203
 ***************Get the Jacobian***************

  0.067703     0.017641   -0.0690447    -0.108936    0.0895563            0
-0.128373   -0.0542373     0.212278    -0.185529   -0.0627688            0
        0    0.0746763     0.210422  -0.00898648     0.210259            0
        0    -0.950962     0.950962     0.019435     0.505894    -0.775423
        0    -0.309307     0.309307    -0.059753     0.861585     0.433043
        1 -4.83063e-17  1.84737e-17     0.998024    0.0417327     0.459556
tiago@tiago-X550JX:~$
```

Figura 3.25: Obtenção das cinemáticas e do jacobinano para uma posição aleatória do robô.

3.4.6 Verificação de colisões

A verificação de colisões no MoveIt! está presente no *planning scene monitor* usando o *CollisionWorld object*. O *CollisionWorld object* suporta diferentes tipos de objetos como por exemplo objetos modelados (os elos de um robô), objetos primitivos (caixas, cones, planos, esferas e cilindros) e do *octomap*. Esta verificação é de tal forma transparente para os utilizadores que estes não necessitam de se preocupar com o seu uso nem com o seu procedimento interno.

Um dos conceitos mais importantes a reter é a matriz *Allowed Collision Matrix* (ACM). Semelhante ao planeamento de uma trajetória, verificar se existem colisões tem uma exigência computacional intensiva e que, muitas vezes, corresponde a cerca de 90% do esforço computacional durante o cálculo de uma trajetória. De forma a facilitar a verificação, simplificam-se os valores da matriz ACM. À posição da matriz relativa a elos em que existe a certeza que nunca colidem, como por exemplo, o elo 1 e 3 do robô Fanuc LR Mate 200iD, é atribuído o número 1. Normalmente esta matriz é construída no assistente de configuração [28].

3.4.7 Ferramenta assistente de configuração

O MoveIt! proporciona uma interface gráfica de forma a integrar novos braços robóticos em ambiente ROS. Esta interface responsabiliza-se por gerar todos os arquivos de configuração e *scripts* de inicialização com base nas informações fornecidas pelo utilizador. Em geral, este é o ponto de partida na utilização do MoveIt!, pois é nesta ferramenta

que são gerados os ficheiros padrão.[17]

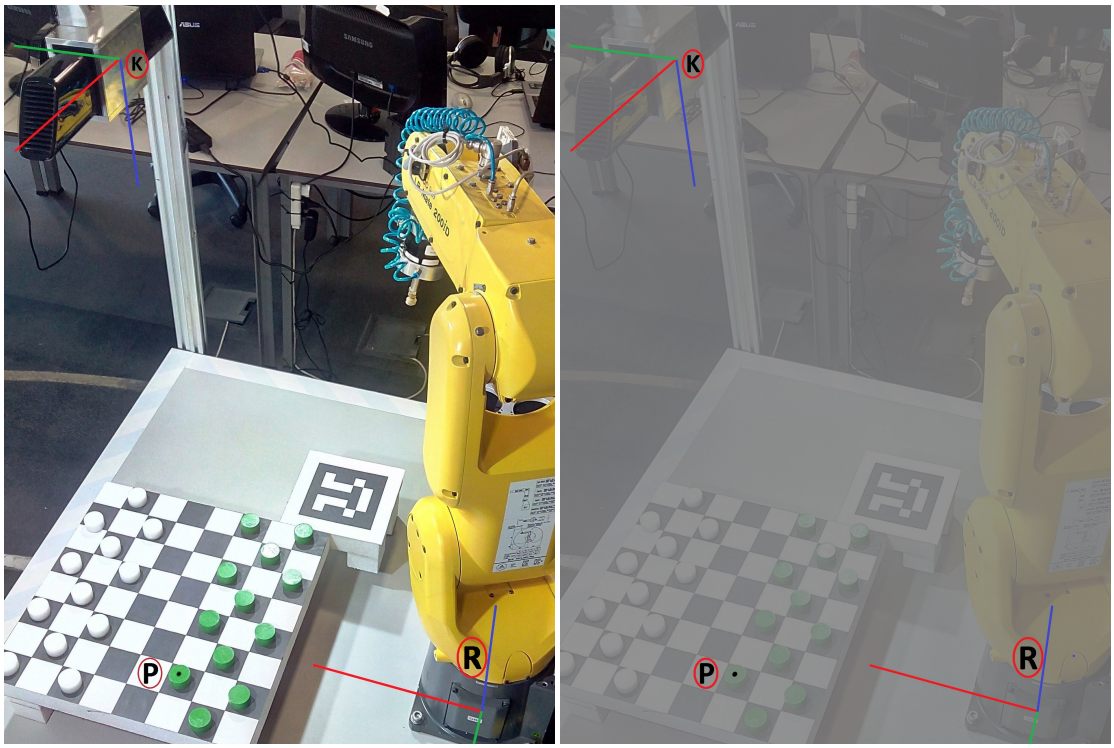
A *Metapackage* ROS Fanuc já continha a pasta dos ficheiros de configuração. No entanto, com a atualização e inclusão do elo do *end-effector*, a atualização dos ficheiros de configuração originais era inevitável. Esta atualização permitiu não só incluir o novo elemento mas também atualizar o TCP. Desta forma, no apêndice A será descrita a metodologia padrão desta ferramenta, já adaptada para o manipulador do presente trabalho.

Capítulo 4

Calibrações

Este capítulo descreve os processos de calibração usados no trabalho. As calibrações determinam as transformações geométricas entre dois sistemas de coordenadas, ou seja, uma relação geométrica de posição e orientação entre os sistemas de coordenadas. Os sistemas de coordenadas utilizados são ortogonais e diretos.

De forma a calcular as relações, é necessário atribuir os sistemas de coordenadas do sistema (figura 4.1a). A figura 4.1b clarifica a representação das posições e as orientações dos sistemas de coordenadas.



(a) Sistemas de coordenadas da base do robô, sensor e de uma peça.

(b) Representação focada nos sistemas de coordenadas.

Figura 4.1: Posicionamento dos principais sistemas de coordenadas.

Os sistemas de coordenadas atribuídos aos elementos do trabalho são:

- R - Sistema de coordenadas da base do robô (`base_link`), localizado da base do robô na mesa de trabalho, referência absoluta;
- K - Sistema de coordenadas do sensor Kinect;
- P - Sistema de coordenadas da superfície do objeto, dependente da localização do objeto;
- H - Sistema de coordenadas do *end-effector* (`tool0`), dependente do posicionamento das juntas do manipulador;
- A - Sistema de coordenadas do *aruco marker*;

O principal objetivo a atingir neste capítulo é calcular as transformações geométricas entre todos sistemas de coordenadas de forma a determinar as coordenadas de posição de uma peça (P), presente no tabuleiro de xadrez, em relação ao sistema de coordenadas do robô (R).

Tomemos como exemplo a simples situação em que se pretendia mover uma peça que estava sempre numa posição fixa. Bastaria medir a distância do seu centro (P) em relação ao sistema de coordenadas da base do robô (R) e enviar as coordenadas para o controlador. Na situação do trabalho, o cálculo da posição da peça em relação ao sistema de coordenadas da base do robô revela-se mais complexa, uma vez que tanto a posição das peças como o tabuleiro são dinâmicas ao longo do jogo de damas. Assim sendo, é essencial determinar todas as transformações geométricas entre os sistemas de coordenadas para que o robô mova uma peça.

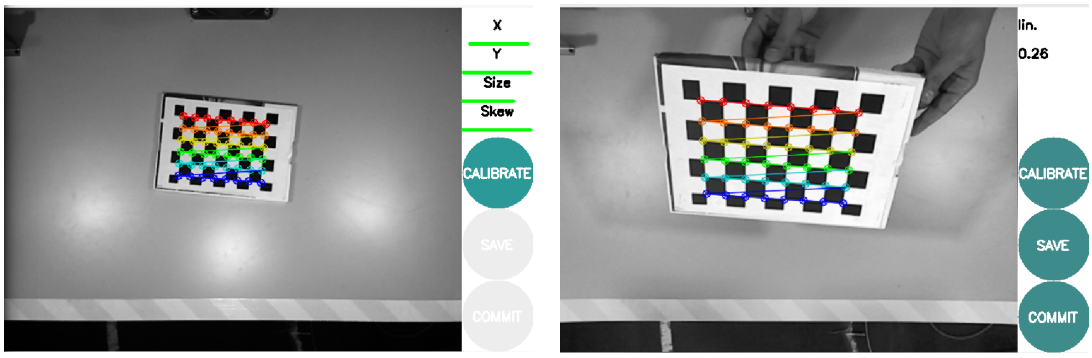
4.1 Determinação dos parâmetros intrínsecos da câmara

Para que a informação obtida pelo sensor seja correta, calibram-se os seus parâmetros. Os parâmetros intrínsecos relacionam as coordenadas em pixel, medidas nas imagens, com as coordenadas de pontos do espaço medidos no sistema de coordenadas com origem no centro do sensor.

Estes parâmetros dependem exclusivamente das características físicas do sensor, da geometria interna e do tipo de lente. Para calibrar os parâmetros intrínsecos, tais como distância focal e ponto focal recorreu-se a um algoritmo de calibração de OpenCV. São conhecidas as dimensões de uma grelha e o programa determina a posição exata no espaço tridimensional de cada ponto da grelha. Este método iterativo ajusta os parâmetros intrínsecos do sensor de modo a minimizar o erro de projeção de cada uma das imagens.

No trabalho utilizou-se um xadrez de 8x6 com 28 mm de espaçamento de cada quadrado. O algoritmo permite a calibração dos parâmetros do sensor RGB e a profundidade. Na figura 4.2a está um exemplo do método de calibração. Através das barras no canto superior direito da figura 4.2a, o programa indica que o número de amostras adquiridas garantem boa qualidade dos quatro parâmetros pretendidos. Por outro lado, a figura 4.2b apresenta o resultado da interface após o cálculo das matrizes resultantes da calibração.

Depois de mover o tabuleiro de xadrez, procede-se à obtenção dos parâmetros intrínsecos, de onde são extraídas as quatro matrizes assinaladas na figura 4.3. São obtidas as matrizes da câmara, dos coeficientes de distorção, de retificação e de projeção. Estas



(a) Durante a calibração. (b) Após a calibração.

Figura 4.2: Metodologia adotada da obtenção dos parâmetros intrínsecos da câmara.

matrizes ficam guardadas no nodo ROS das informações do sensor e em disco. No decorrer do trabalho utilizou-se a matriz 3x3 da câmara, de modo a transformar os dados de uma imagem com coordenadas em pixels para coordenadas do mundo.

```
image_width: 640
image_height: 480
camera_name: depth_A00367A02265044A
camera_matrix:
  rows: 3
  cols: 3
  data: [592.7066538668316, 0, 313.7151676040007, 0, 592.8470212888272, 237.4767429148477, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [-0.04489643566566365, 0.1522174888837654, -0.009326188734865081, -0.0006381285081139927, 0]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
  rows: 3
  cols: 4
  data: [601.5407104492188, 0, 313.1500592708799, 0, 0, 597.5403442382812, 232.9643824233081, 0, 0, 0, 1, 0]
```

Figura 4.3: Obtenção das quatro matrizes dos parâmetros intrínsecos da câmara.

4.2 Determinação dos parâmetros extrínsecos da câmara

Os parâmetros extrínsecos da câmara referem-se à posição e na orientação do sistema de coordenadas do sensor relativamente a outro sistema de coordenadas. A determinação dos parâmetros extrínsecos nem sempre é necessária nas situações em que se pretende utilizar exclusivamente o sistema de coordenadas da câmara. No trabalho em causa, foi imprescindível o recurso aos parâmetros extrínsecos, uma vez que se calcularam as relações entre sistemas de coordenadas e obtiveram-se as transformações entre eles. Um caso específico, que será estudado mais adiante, é o cálculo da transformação dos sistemas de coordenadas do sensor em relação ao sistema de coordenadas de um marcador especial.

Uma matriz de rotação (R) é uma matriz de dimensão 3x3 e pode ser determinada a partir do conhecimendo dos ângulos. O sistema de coordenadas de referência tem que girar em torno de cada um do seus eixos para se transformar num novo sistema de

coordenadas. Caso se pretenda alterar a posição do sistema de coordenadas, recorre-se a uma matriz de translação (P) que é um vetor coluna de dimensão 3x1 [36].

$$R = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Uma matriz de transformação transforma um ponto noutro. De forma a melhorar a performance computacional, recorreu-se à solução de coordenadas homogéneas que proporciona a construção mais compacta da matriz de transformação (T).

$$T = \left[\begin{array}{ccc|c} a & b & c & p_x \\ d & e & f & p_y \\ g & h & i & p_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

A rotação pura pode ser feita em torno de cada um dos seus eixos (x,y,z). Assim, a rotação em torno de cada um dos seus eixos é dada por três matrizes.

$$Rot(x,\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad Rot(y,\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

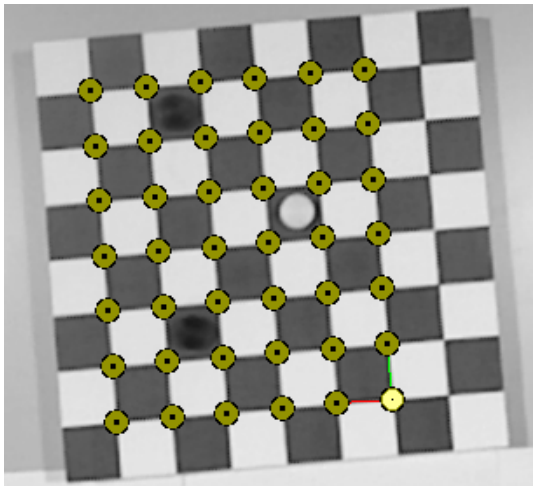
$$Rot(z,\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

No caso de rotações compostas em torno de dois ou mais eixos, a componente de rotação da matriz de transformação será mais complexa.

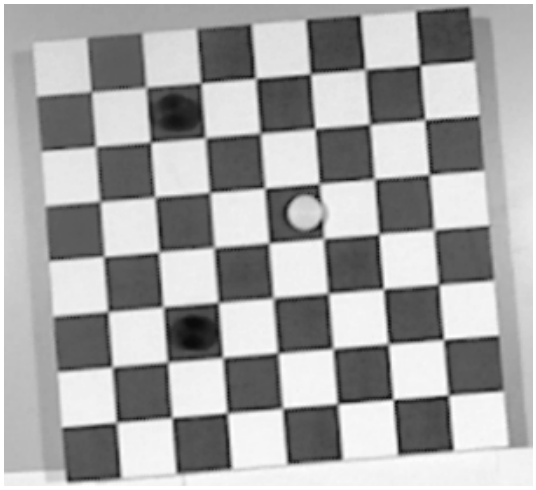
Como o tabuleiro de jogo tem um padrão similar a um tabuleiro de xadrez, optou-se por um algoritmo que reconhecesse esse padrão, de forma a calcular os parâmetros extrínsecos (figura 4.4a). Esta metodologia era válida e funcional até ao momento em que o posicionamento de uma das peças fizesse com que uma das linhas não fosse visível. Sem o padrão completo do xadrez, o algoritmo deixava de detetar o tabuleiro e não calculava a transformação geométrica. Por exemplo, um simples mau posicionamento da peça branca dentro do quadrado da figura 4.4b fez com que o padrão deixasse de ser reconhecido.

Outro ponto a salientar nos algoritmos desta natureza é a necessidade de um padrão retangular de modo a definir uma origem do sistema de coordenadas atribuído ao xadrez. Como o tabuleiro tem um padrão quadrado, verificou-se uma ambiguidade na localização da origem do sistema de coordenadas.

A solução encontrada passou por definir um sistema de coordenadas que fosse independente do padrão do tabuleiro. Desta forma, recorreu-se a uma *package* ROS que contém algoritmos que fazem o reconhecimento de um marcador especial (figura 4.5). A *package* em causa intitula-se de *ARUCO / VISP Hand-Eye Calibration* [37] e utiliza um



(a) Identificação correta.



(b) Falha na identificação.

Figura 4.4: Representação da identificação dos padrão do xadrez.

alvo (*aruco marker*) e a calibração (VISP) que faz uma estimativa da posição do sistema de coordenadas do sensor. O algoritmo possibilita a escolha de duas estimativas possíveis, *eye-in-hand*/*eye-on-base*. A estimativa adotada foi a *eye-on-base*. Nesta situação, o sensor é fixo numa posição no ambiente do robô. O *aruco marker* escolhido foi o 582 [38] (figura 4.5) quadrado e com 123 mm de lado. No entanto, para que o *aruco marker* seja reconhecido pelo algoritmo, é necessário um bordo branco. Para o trabalho, o bordo tinha 25 mm, ficando com a dimensão total final de 173 mm de lado.

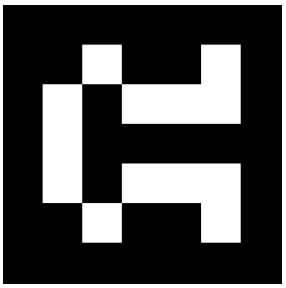
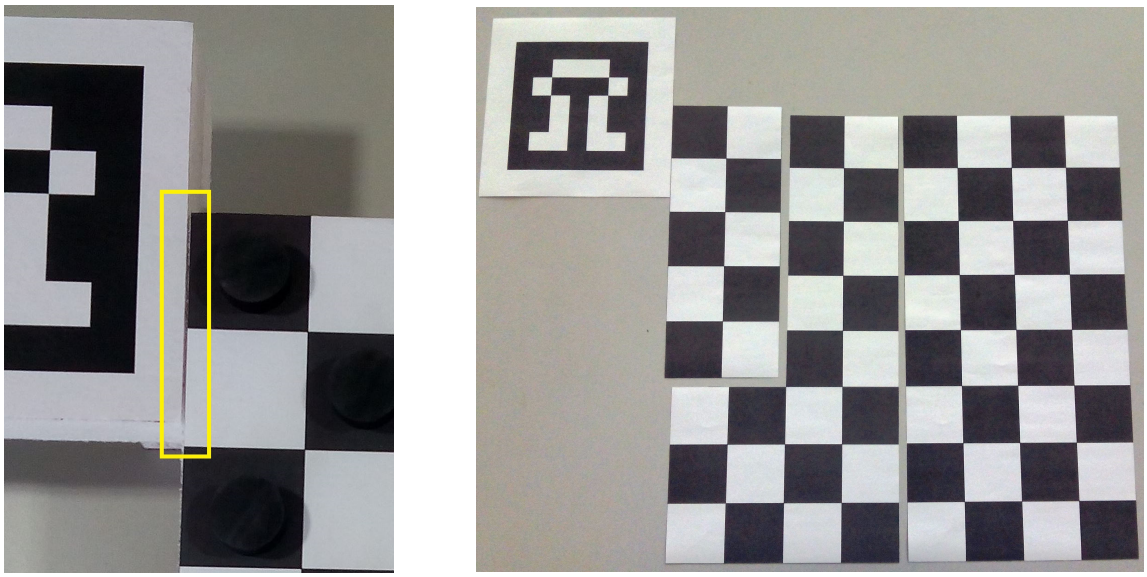


Figura 4.5: *Aruco marker* 582 [38].

Ao adaptar o tabuleiro de forma a incluir o *aruco marker*, verificou-se que a união não ficou perfeita (figura 4.6a). Esta imperfeição não garantia um perfeito paralelismo dos eixos do sistema de coordenadas do *aruco marker* com as linhas e as colunas do tabuleiro. Com este desalinhamento, a distância das linhas e colunas a cada um dos eixos variava à medida que se afastava da origem do *aruco marker*.

De forma a corrigir o desalinhamento, construiu-se um segundo tabuleiro de xadrez (figura 4.6b). Este novo tabuleiro (figura 4.7) foi construído com o *aruco marker* anexado, garantindo que as linhas e as colunas ficassem paralelas com o sistema de coordenadas gerado no *aruco marker*.

Em ambiente ROS, a transformação entre os dois sistemas de coordenadas é conse-



(a) Detalhe da origem do desvio. (b) Correção da falha.

Figura 4.6: Correção da união entre o *aruco marker* e o tabuleiro de xadrez.

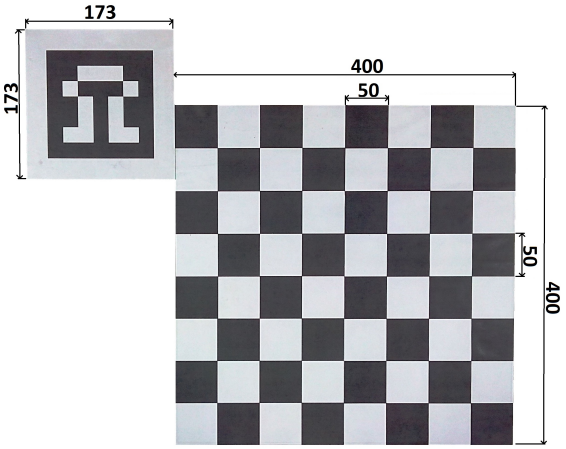


Figura 4.7: Dimensões do tabuleiro de xadrez com o *aruco marker*.

guida sob a forma de uma mensagem TF (figura 4.8) que contém a informação que se presente adquirir. A transformação é constituída pela translação, em metros, e a rotação, em radianos. A componente da rotação é dada em formato de *quaternion*.

```
header:
  seq: 12517
  stamp:
    secs: 1467824351
    nsecs: 615816216
  frame_id: /camera_rgb_optical_frame
child_frame_id: /hand_eye//camera/rgb/aruco_marker_frame
transform:
  translation:
    x: -0.00610343972221
    y: 0.00120460358448
    z: 0.253400921822
  rotation:
    x: -0.503193745883
    y: 0.498353165153
    z: -0.497566711667
    w: 0.500866801013
```

Figura 4.8: Mensagem ROS relativa à transformação entre os sistemas de coordenadas do sensor e do *aruco marker*.

4.3 Grafos e equações de transformação

Para diversos problemas em robótica industrial, é necessário determinar a posição relativa entre os componentes em relação a um sistema de coordenadas comum. No nosso caso, é imprescindível estabelecer a relação entre todos os sistemas de coordenadas de forma a que o robô tenha movimentos condicionados.

O sistema real da figura 4.14a representa a disposição dos componentes e tem ilustrado os sistemas de coordenadas. A figura 4.14b representa e clarifica a posição e orientação de cada um dos sistemas de coordenadas.

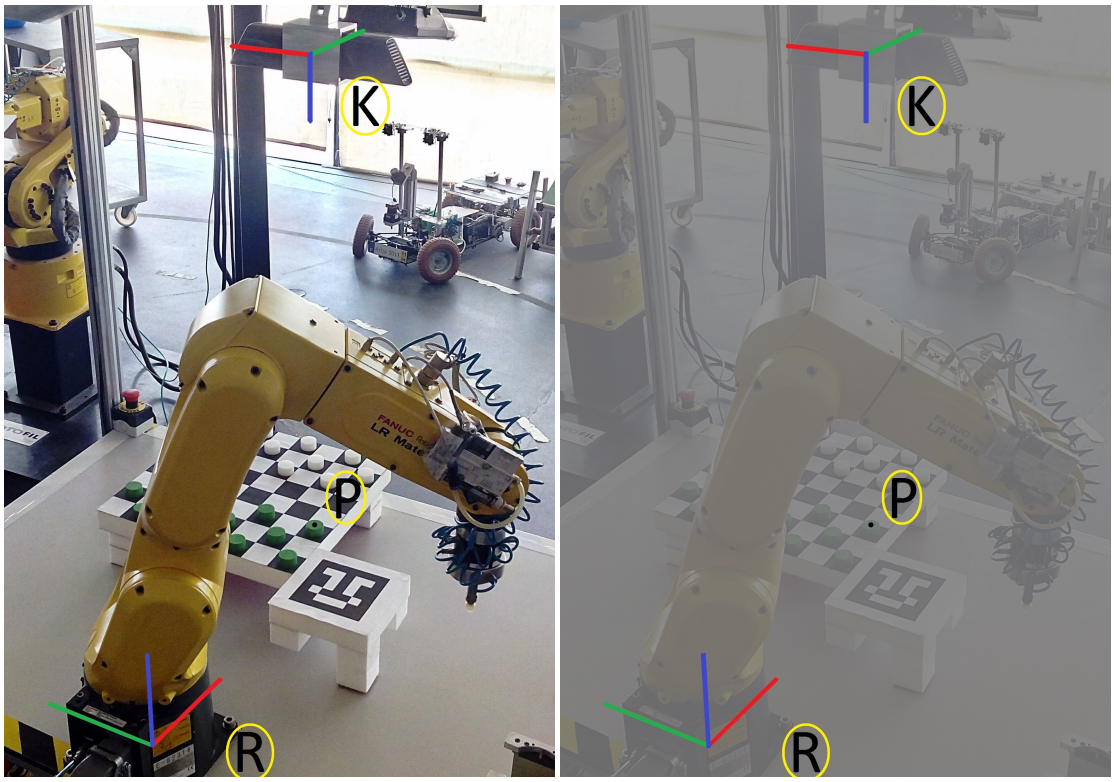
Para calcular a transformação geométrica entre as coordenadas do ponto (P) e o sistema de coordenadas do robô (R) construiu-se o grafo de transformações (figura 4.10). Este é o principal grafo de transformações.

Do grafo da figura 4.10 resulta a equação 4.1

$$^{R}T_{P} = {}^{R}T_{K} \times {}^{K}T_{P} \tag{4.1}$$

A equação 4.1 é composta por três termos que são descritos de seguida:

- $^{R}T_{P}$ - Transformação geométrica entre o sistema de coordenadas da base do robô e as coordenadas do centro geométrico da superfície do objeto a manipular;
- $^{R}T_{K}$ - Transformação geométrica entre o sistema de coordenadas da base do robô e o sistema de coordenadas do sensor Kinect;
- $^{K}T_{P}$ - Transformação geométrica entre o sistema de coordenadas do sensor e as coordenadas do centro geométrico da superfície do objeto a manipular;



(a) Representação real incluindo os sistemas de coordenadas.

(b) Representação focada nos sistemas de coordenadas.

Figura 4.9: Disposição dos componentes e representação dos sistemas de coordenadas.

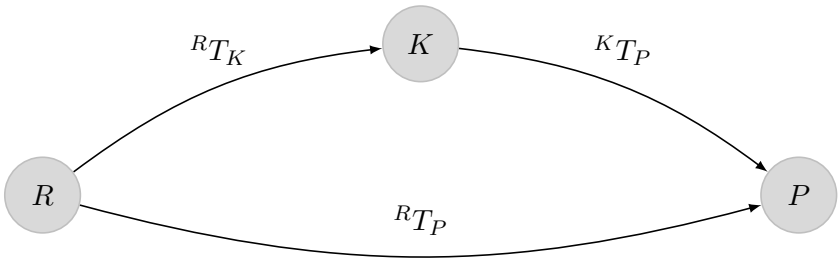


Figura 4.10: Grafo de transformações que relaciona os sistemas de coordenadas da base do robô, peça e sensor.

Para auxiliar o cálculo dos termos ${}^{R}T_{K}$ e ${}^{K}T_{P}$, foram criados dois grafos de transformações auxiliares. O primeiro grafo de transformações (figura 4.11) permite o cálculo do termo ${}^{R}T_{K}$, enquanto que o segundo grafo de transformações (figura 4.13) permite o cálculo do termo ${}^{K}T_{P}$.

4.3.1 Cálculo da transformação geométrica ${}^{R}T_{K}$

Para o cálculo do primeiro termo, o ${}^{R}T_{K}$, recorre-se ao primeiro grafo auxiliar (figura 4.11) de onde resulta a equação 4.2.

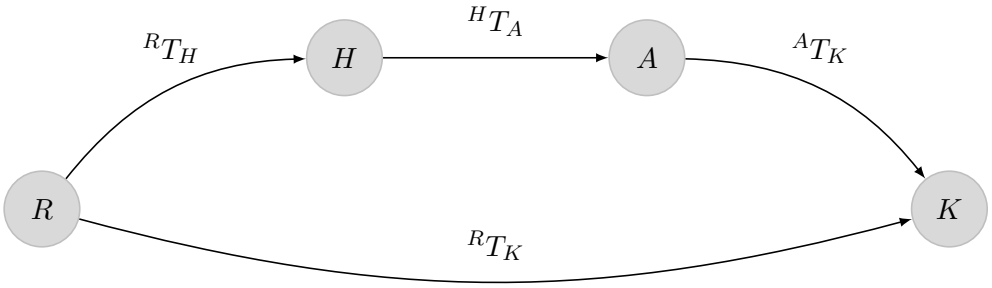


Figura 4.11: Grafo de transformações que relaciona os sistemas de coordenadas da base do robô, *end-effector*, *aruco marker* e sensor.

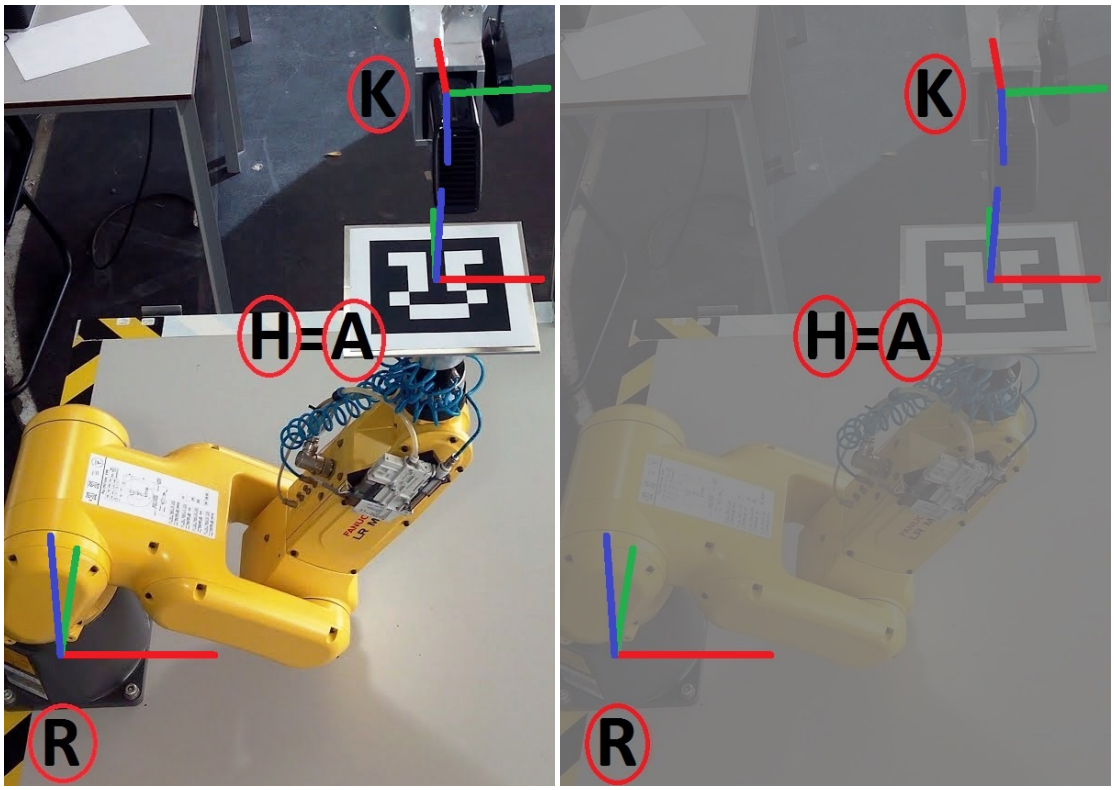
$$ {}^{R}T_{K} = {}^{R}T_{H} \times {}^{H}T_{A} \times {}^{A}T_{K} \tag{4.2} $$

A equação 4.2 é composta por quatro termos que são descritos de seguida:

- ${}^{R}T_{K}$ - Transformação geométrica entre o sistema de coordenadas da base do robô e o sistema de coordenadas do sensor Kinect;
- ${}^{R}T_{H}$ - A determinação desta transformação geométrica provem das transformações (TF) do ROS. Estas calculam a transformação instantânea entre dois sistemas de coordenadas pretendidos. Neste caso, pretendia-se que os dois sistemas de coordenadas fossem `base_link` e `tool0`, que correspondem ao sistema de coordenadas da base do robô e do *end-effector*, respetivamente;
- ${}^{H}T_{A}$ - A matriz de transformação entre o sistema de coordenadas do *end-effector* e do *aruco marker* (${}^{H}T_{A}$) deve corresponder a uma matriz de identidade na componente de rotação e zero na translação. De forma a coincidir, em posição e orientação, os dois sistemas de coordenadas do *end-effector* e o *aruco marker*, projetou-se e desenvolveu-se uma ferramenta de calibração;
- ${}^{A}T_{K}$ - A determinação desta transformação está diretamente relacionada com a *package ARUCO / VISP Hand-Eye Calibration* e dependente da posição do *aruco marker*. A transformação obtém-se diretamente desta *package* e tem uma estrutura similar à figura 4.8;

O resultado da equação 4.2 é constante e corresponde unicamente à transformação entre os dois sistemas de coordenadas principais, base do robô e sensor. Caso o sensor Kinect ou a base do manipulador mudem de posição ou de orientação, a matriz de

transformação entre os dois sofrerá variações nos seus valores, pelo que necessita de ser recalculada. O cálculo da transformação geométrica ${}^{R}T_{K}$ envolve tarefas, tais como posicionar o braço robótico, acoplar a ferramenta de calibração, posicionar o *aruco marker* no topo da ferramenta de calibração e calcular as transformações entre os quatro sistemas de coordenadas (figura 4.12a). A figura 4.12b clarifica a representação das posições e orientações dos sistemas de coordenadas.



(a) Representação real incluindo os sistemas de coordenadas.

(b) Representação focada nos sistemas de coordenadas.

Figura 4.12: Imagem real da posição do robô na posição de calibração, incluindo a representação focada nos sistemas de coordenadas.

De forma a facilitar todo o processo, desenvolveu-se uma metodologia automática para calcular a matriz de transformação ${}^{R}T_{K}$. A explicação do cálculo e do algoritmo que lhe deu origem está descrita na secção 4.3.4. Com isto, ${}^{R}T_{K}$ está determinado.

4.3.2 Cálculo da transformação geométrica ${}^{K}T_{P}$

Para calcular o segundo e último termo da equação 4.1, apresenta-se o segundo grafo de transformações auxiliar (figura 4.13).

Da análise do grafo da figura 4.13, resulta a equação 4.3

$$ {}^{A}T_{P} = {}^{A}T_{K} \times {}^{K}T_{P} \iff {}^{K}T_{P} = {}^{K}T_{A} \times {}^{A}T_{P} \tag{4.3} $$

A equação 4.3 é composta por três termos que são descritos de seguida:

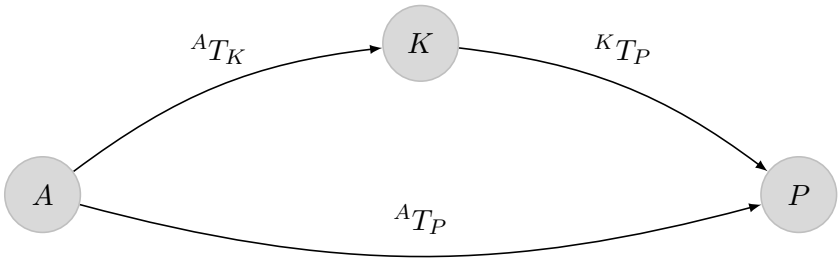
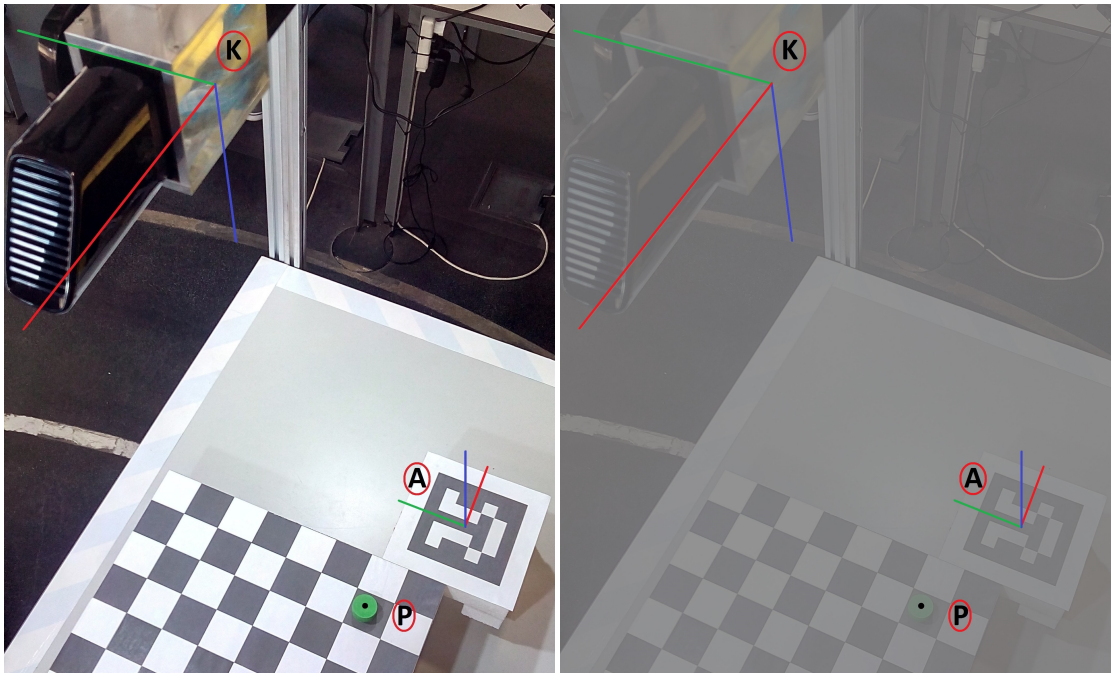


Figura 4.13: Grafo de transformações que relaciona os sistemas de coordenadas do *aruco marker*, sensor e da peça.

- $^{A}T_{P}$ - Transformação geométrica do sistema de coordenadas do *aruco marker* para o centro geométrico da superfície de todas as peças presentes no tabuleiro;
- $^{K}T_{A}$ - A determinação desta transformação está diretamente relacionada com a *package ARUCO / VISP Hand-Eye Calibration* e dependente da posição do *aruco marker*. A transformação obtém-se diretamente desta *package* e tem uma estrutura similar à figura 4.8;
- $^{K}T_{P}$ - Transformação geométrica do sistema de coordenadas do sensor para as coordenadas do centro geométrico da superfície da peça a mover na jogada do robô;



(a) Representação real incluindo os sistemas de coordenadas.

(b) Representação focada nos sistemas de coordenadas.

Figura 4.14: Disposição dos componentes e representação dos sistemas de coordenadas.

A equação 4.3 é composta por duas equações.

A primeira equação, ${}^{A}T_P = {}^{A}T_K \times {}^{K}T_P$, pretende atribuir um sistema de coordenadas comum (o *aruco marker*) para todas as peças presentes no tabuleiro. Isto é, calcula a posição de cada peça em relação ao sistema de coordenadas do *aruco marker*, passando pelo sistema de coordenadas do sensor Kinect. A posição de cada peça é obtida por técnicas de visão artificial (o ${}^{K}T_P$). Por exemplo, na situação inicial de um jogo de damas, o tabuleiro é composto por 24 peças, o que quer dizer que esta equação é executada 24 vezes.

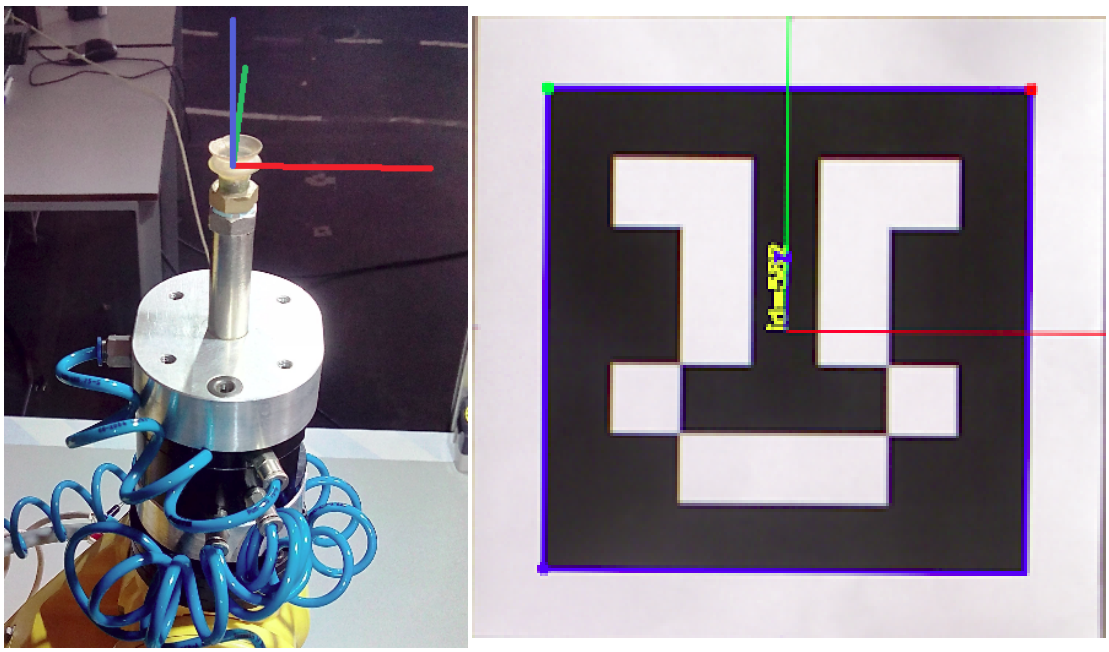
A segunda equação, ${}^{K}T_P = {}^{K}T_A \times {}^{A}T_P$, pretende determinar a posição da peça a mover na jogada do robô. Após o algoritmo de jogo determinar a peça que o robô terá de mover, a posição da peça é convertida do sistema de coordenadas do *aruco marker* para o sistema de coordenadas do sensor Kinect.

Com isto, ${}^{K}T_P$ está determinado e a equação 4.1 completa.

O cálculo e o desenvolvimento da programação onde estão inserias as duas transformações geométricas referenciadas anteriormente estão descritas na secção 5.2.2 e na secção 6.2.

4.3.3 Desenvolvimento da ferramenta de calibração

A calibração do sensor Kinect com o manipulador é um processo suscetível a erros, mais propriamente na questão do alinhamento e da coincidência da origem do sistema de coordenadas do TCP do robô (figura 4.15a) com o sistema de coordenadas do *aruco marker* (figura 4.15b). A posição do TCP do robô situa-se na extremidade do *end-effector*.



(a) Sistema de coordenadas TCP do robô.

(b) Posição e orientação do sistema de coordenadas do *aruco marker*.

Figura 4.15: Localização 3D do sistema de coordenadas TCP do robô e o *aruco marker*.

Dada a importância deste procedimento, projetou-se e desenvolveu-se um calibrador

para minimizar os erros na obtenção da matriz de transformação da base do manipulador para o sensor. Seguidamente, é apresentada uma vista de conjunto (figura 4.16) e uma imagem real da ferramenta (figura 4.17) onde está presente o *end-effector* e o respetivo suporte de calibração. A sequência de montagem é realizada pela sequência amarela e posteriormente vermelha. Os desenhos de definição do pino, chapa e pino apoio estão presentes no apêndice B.

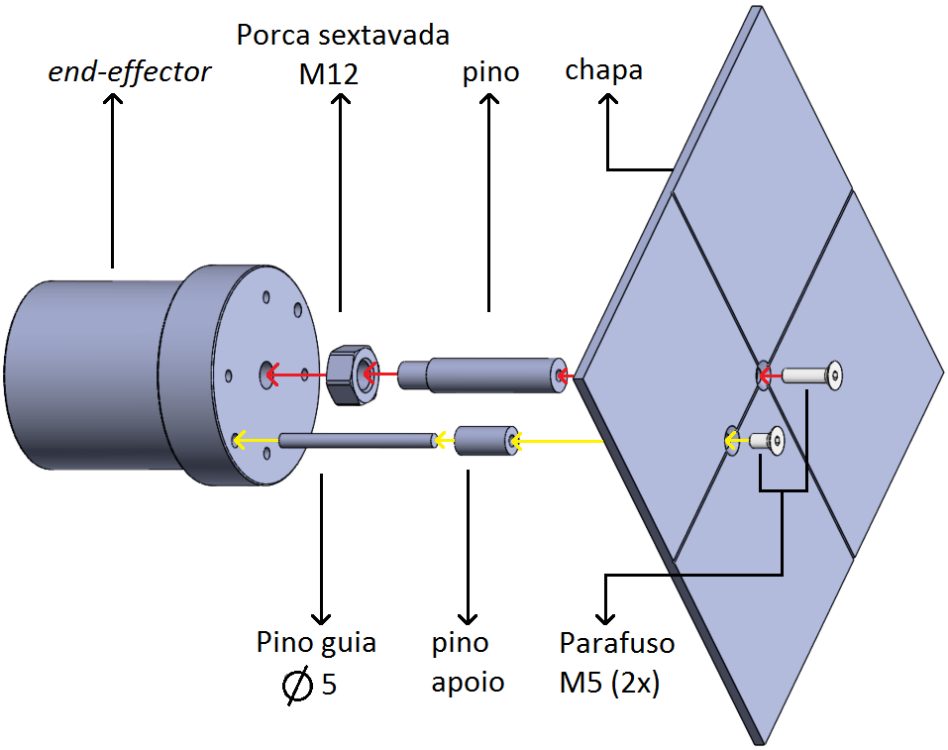


Figura 4.16: Vista explodida do calibrador.

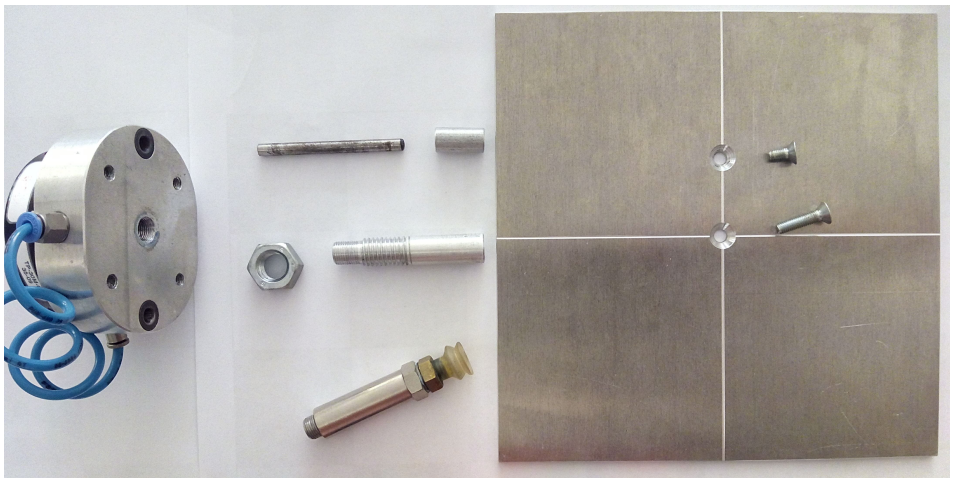
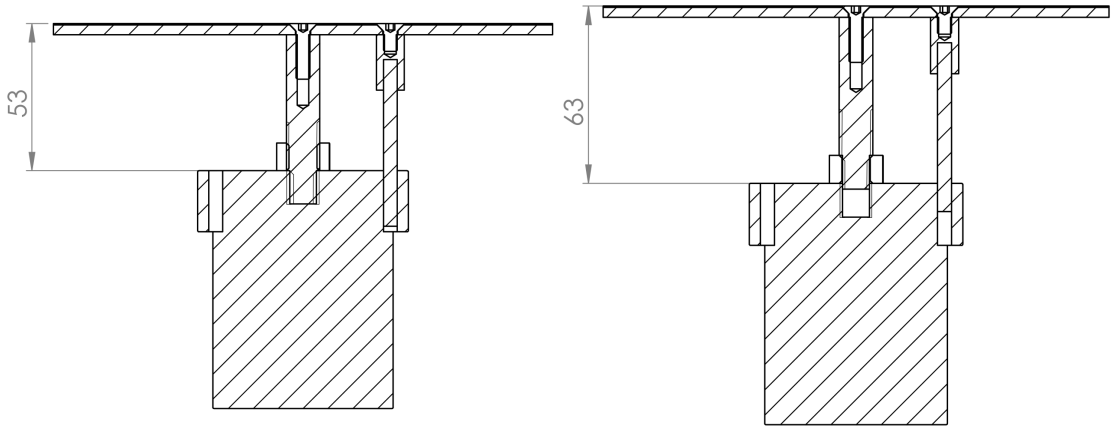


Figura 4.17: Imagem real da vista explodida do calibrador.

O calibrador foi pensado e estruturado para possibilitar ao utilizador diversos ajustamentos. Para uma melhor compreensão da sua funcionalidade e robustez, será apresentado um exemplo. A orientação do sistema de coordenadas do *end-effector* e do *aruco marker* são iguais. Primeiramente, regula-se a cota em z de forma a coincidir as duas origens. Sabe-se que a posição do TCP encontra-se algures no centro da ventosa e centrado com o seu furo (figura 4.15a). Para que a coincidência dos centros fosse o mais rigorosa possível, possibilita-se o ajuste da cota z, mediante o comprimento da rosca do furo presente no *end-effector*. Esta rosca tem um curso de 10 mm e foi projetada de forma a que a posição mínima (figura 4.18a) fosse coincidente com a face da porca, junto à ventosa, e a posição máxima (figura 4.18b) fosse coincidente com a parte superior da ventosa.



(a) Posição mínima.

(b) Posição máxima.

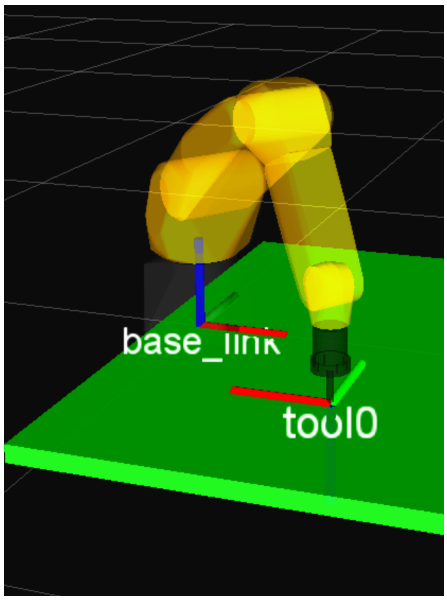
Figura 4.18: Representação das posições mínima e máxima que o calibrador ocupa.

Em ROS-Industrial, a atribuição da localização da origem do TCP do modelo é realizada no ficheiro `lrmate200id.urdf` contido na *package* `fanuc_lrmate200id_support`, pasta `urdf` do ROS Fanuc. Uma vez que o modelo do *end-effector* foi construído a partir da medição da ferramenta real, não se tinha a garantia que a posição da extremidade do modelo correspondesse à posição da extremidade real.

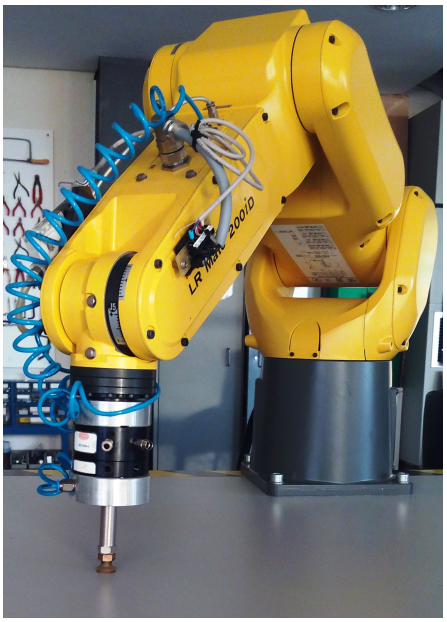
De modo a garantir que a posição do TCP atribuída no ROS fosse coincidente com a posição real do TCP do *end-effector*, movimentou-se o braço robótico para uma posição conhecida no plano e com a coordenada z igual a 0, para a coincidir com a superfície da mesa de trabalho. Na figuras 4.19a e 4.19b verifica-se que as posições do TCP são coincidentes.

Com a garantia que a posição do TCP em ROS e real coincidiam, movimentou-se o manipulador para a posição de segurança onde foi anexado o calibrador ao *end-effector* na posição mínima. Com o calibrador no *end-effector*, reposicionou-se o braço robótico para a posição de coincidência com a superfície da mesa de trabalho (figura 4.20). Desta forma, garantiu-se a perfeita coincidência da posição dos sistemas de coordenadas do calibrador e do TCP do robô. A presença da porca garantiu que a posição fosse mantida a partir daquele momento.

Tendo a coincidência, em posição, dos dois sistemas de coordenadas garantida, levanta-se a questão da coincidência das orientações. Tal problema foi solucionado não só pela



(a) Posição do TCP em ROS-Industrial.



(b) Posição do TCP do robô.

Figura 4.19: Comparação do valor do TCP do modelo em ROS-Industrial e real.

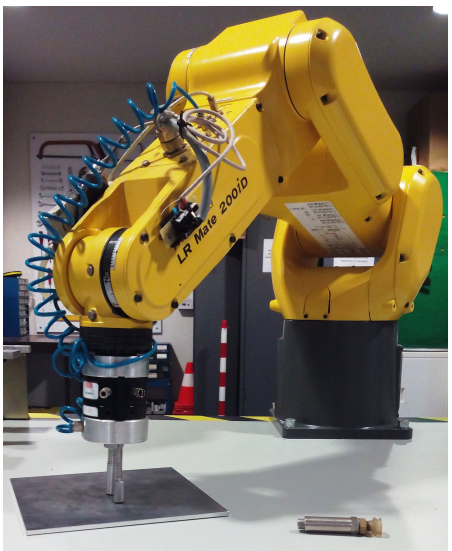
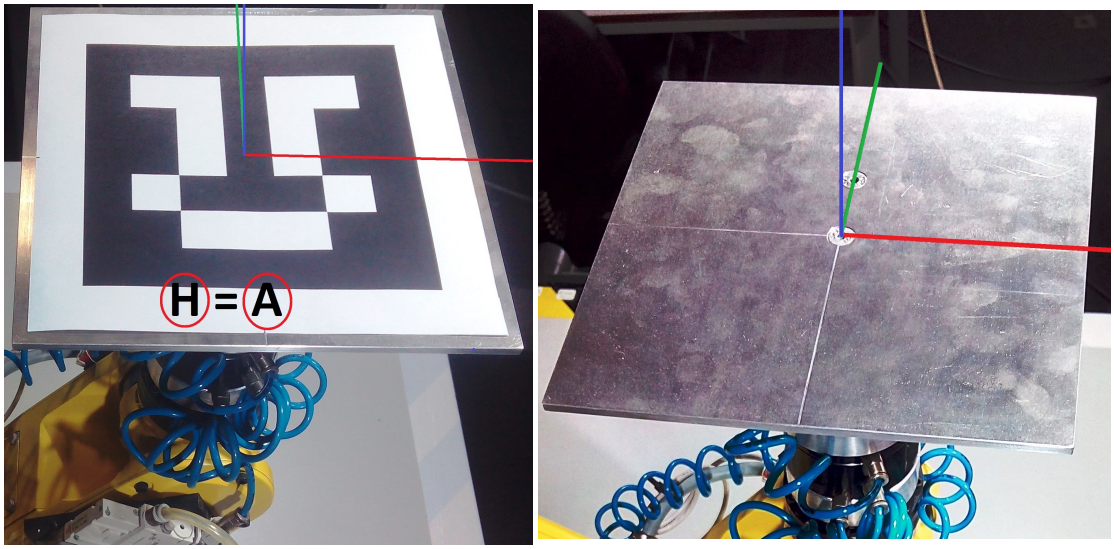


Figura 4.20: Ajuste da cota z do calibrador.

presença de um segundo ponto de fixação, mas também pela presença do parafuso no eixo do furo central. A existência de um segundo ponto de fixação garantiu que a rotação em torno do eixo z fosse bloqueada. Já a existência do parafuso assegurou que a chapa quadrada ficasse perfeitamente perpendicular ao veio central.

No fim deste procedimento garante-se que o calibrador fica completamente imóvel e coincidente com o TCP do robô. A figura 4.21a ilustra a coincidência dos dois sistemas de coordenadas. Outra questão que surgiu foi o posicionamento da folha do *aruco marker* na superfície do calibrador. Tal questão foi ultrapassada com a gravação de duas ranhuras na superfície da chapa, perfeitamente coincidentes com o centro do veio figura 4.21b. Desta forma, promoveu-se a coincidência do centro da folha com o centro da chapa.



(a) Sistemas de coordenadas coincidentes.

(b) Representação das ranhuras.

Figura 4.21: Posicionamento do *aruco marker* na superfície do calibrador e coincidência dos sistemas de coordenadas.

4.3.4 Calibração automática

Foram desenvolvidos dois nodos ROS de forma a automatizar o processo de calibração e resolver a equação 4.2 de forma automática. Os dois nodos intitulam-se `posicao_de_calibracao` e `matrix_RoboToCam`.

4.3.4.1 Nodo `posicao_de_calibracao`

O primeiro nodo responsabiliza-se unicamente por movimentar o manipulador para três posições específicas. As posições foram desenvolvidas para permitir que o utilizador, através de uma única linha de código, posicione o robô numa das três posições: *home position*, na posição do TCP (figura 4.19b) e na posição de calibração (figura 4.12). Na figura 4.22 é descrita a metodologia adotada para este nodo.

Conforme o esquema presente na figura 4.22, o nodo começa com a construção e o posicionamento dos modelos dos objetos presentes no ambiente do robô. A presença destes objetos é importante para que o MoveIt! calcule as trajetórias sem colisões ou

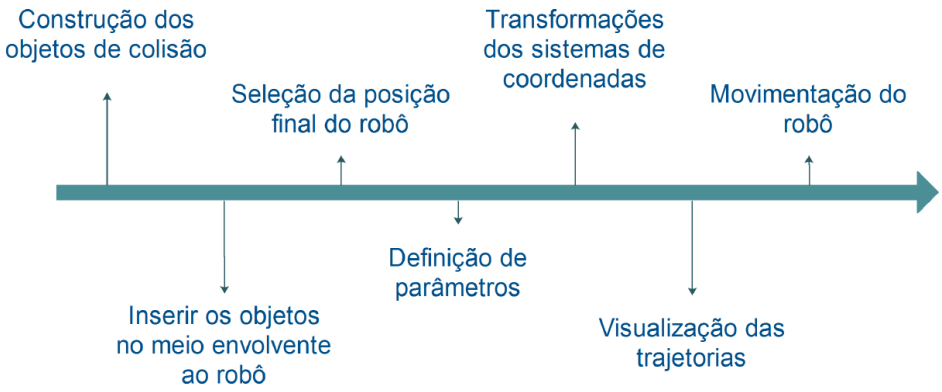


Figura 4.22: Sequência temporal dos acontecimentos do nodo ROS `posicao_de_calibracao`.

interferências com o robô. Os principais objetos criados foram a geometria do sensor Kinect, a estrutura de fixação e a mesa de trabalho, tudo à escala real (figura 4.23).

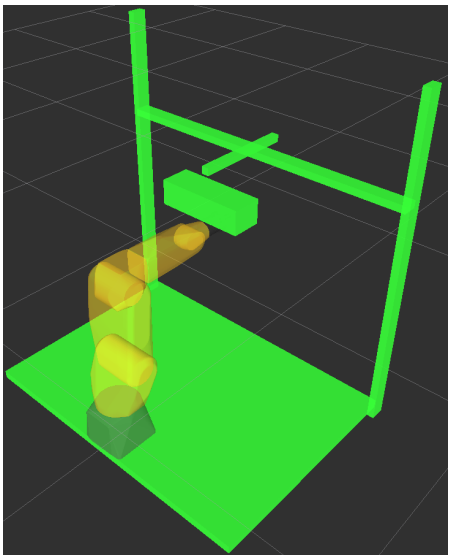


Figura 4.23: Posicionamento dos elementos representativos do ambiente envolvente com o robô na *home position* no simulador (RViz).

Seguidamente, dá-se a possibilidade ao programador de selecionar uma das três posições disponíveis. Estas utilizam três funções respeitantes a três posições diferentes que são descritas de seguida:

- `posicao_calibracao(group,0.480,0.000,0.680)` - Função que recebe como argumentos a classe `move_group` e as três coordenadas de posição (x,y,z) para mover o robô. A posição em causa é a posição de calibração (figuras 4.24a e 4.25);
- `posicao_tcp(group,0.400,-0.300,0.000)` - Função que recebe como argumentos

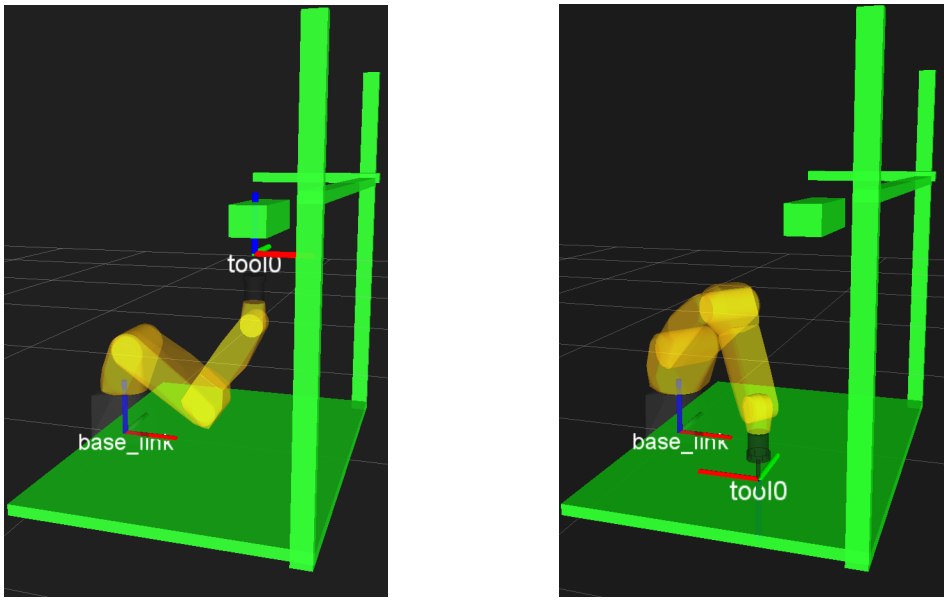
a classe `move_group` e as três coordenadas de posição (x,y,z) para mover o robô. A posição em causa é a posição do TCP (figuras 4.19b e 4.24b);

- `home_position(group)` - Função que recebe como argumento a classe `move_group` e envia o manipulador para a *home position* por movimentos de juntas;

A primeira e segunda funções seguem o modelo da figura 4.22. Primeiramente, definem-se algumas das principais configurações para `move_group`. As configurações apresentadas foram: o tempo de planeamento, a especificação do elo do *end-effector*, o número de tentativas para o planeamento e a tolerância que o *end-effector* terá que garantir para uma posição.

Posteriormente, define-se a orientação do sistema de coordenadas da ferramenta. Esta especificação dá-se com rotações em torno de cada um dos eixos em cada sistema de coordenadas. Por exemplo, na figura 4.24a, na transformação entre o sistema de coordenadas da base do robô e o sistema de coordenadas do *end-effector*, a matriz de rotação é a matriz identidade, ou seja, não existe alteração nas orientações do sistema de coordenadas base para o sistema de coordenadas do *end-effector*. No entanto, as orientações do sistema de coordenadas do *end-effector* precisam de ser modificadas, permitindo o controle da sua direção. Partindo do sistema de coordenadas base do robô (base_link), aplicou-se uma rotação de π radianos em torno do eixo y (verde), fazendo com que o último sistema de coordenadas fosse o representado na figura 4.24b.

De seguida, visualiza-se a trajetória entre a posição corrente do manipulador até à posição pretendida e movimenta-se o braço robótico. O caminho percorrido pelo robô durante as posições é igual ao percurso do simulador RViz.



(a) Sistemas de coordenadas sem rotação. (b) Sistemas de coordenadas com rotação.

Figura 4.24: Representação das transformações geométricas aplicadas ao *end-effector*.

A terceira função, `home_position(group)`, é responsável por enviar o manipulador para a posição *home position*. Esta posição é definida na criação do assistente de configuração. Esta função não segue, na totalidade, o modelo da figura 4.22. Até ao terceiro

passo o procedimento é igual, no entanto diferencia-se a partir do quarto passo, onde é atribuído o valor zero a cada uma das seis juntas. Posteriormente, visualiza-se o trajeto entre a posição corrente do robô e a *home position*. Por fim, envia-se a trajetória para o controlador e executa-se o movimento no robô.

4.3.4.2 Nodo `matrix_RoboToCam`

O segundo nodo ROS, `matrix_RoboToCam`, é executado quando o manipulador se encontra na posição de calibração (figura 4.25) e é neste nodo que é calculada a transformação geométrica ${}^{R}T_{K}$. Na figura 4.26 é apresentada a ordem cronológica das ações e do cálculo da matriz de transformação geométrica.

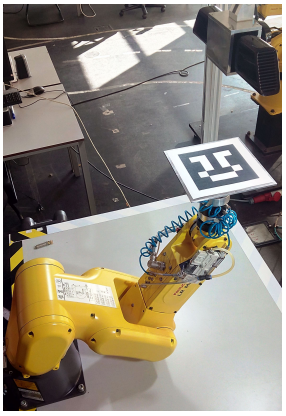


Figura 4.25: Imagem real da posição de calibração.

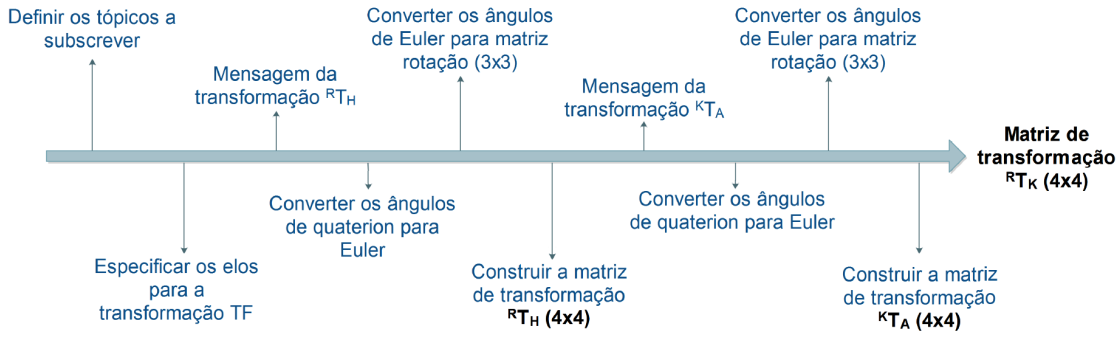


Figura 4.26: Sequência temporal dos acontecimentos do nodo ROS `matrix_RoboToCam` de forma a obter a transformação geométrica ${}^{R}T_{K}$.

O nodo ROS inicia com a especificação dos tópicos que vão ser subscritos para obter as transformações pretendidas. São subscritos três tópicos `/tf`, `/tf_static` e `/camera/rgb/aruco_tracker/transform` (figura 4.27).

A subscrição e relação entre os dois primeiros tópicos possibilita o cálculo da transformação instantânea da base do robô (`base_link`) para o *end-effector* (`tool0`). Após se especificarem os elos para se obter a transformação, recebe-se uma mensagem correspondente à transformação entre eles com informação da posição e orientação, em *quaternion*.

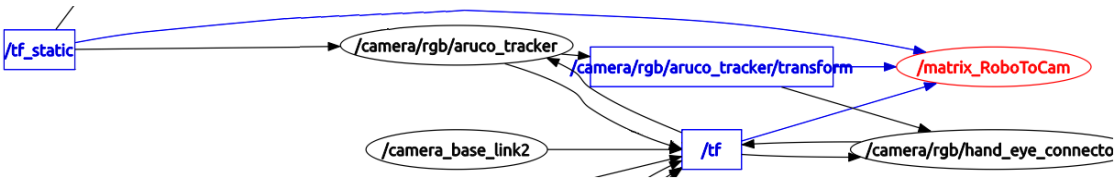


Figura 4.27: Nodo (a vermelho) e tópicos subscritos (a azul).

Como se pretende obter uma matriz transformação 4x4 adotou-se uma sequência de conversões bem específica. A primeira conversão foi passar de *quaternions* para ângulos de Euler. Tendo os ângulos de Euler, converteram-se ângulos de Euler na matriz 3x3 de rotação. Para a construção da matriz 4x4, utilizou-se a matriz de rotação com a especificação da posição proveniente dos *quaternions*.

A figura 4.28 apresenta a mensagem recebida e as conversões descritas anteriormente. Este método é fiável, uma vez que os valores calculados da matriz de transformação ${}^{R}T_{H}$ correspondem aos valores teóricos, isto é, na posição de calibração a matriz de rotação é a matriz identidade e a componente de posição corresponde aos valores da função `posicao_calibracao`.

```
######### MENSAGEM RECEBIDA EM FORMATO QUATERION #########
Posicao x =0.480097
Posicao y =5.46884e-05
Posicao z =0.679959
Orientacao x =-2.00553e-05
Orientacao y =7.67441e-05
Orientacao z =-0.000183663
Orientacao w =1

######### CONVERSÃO DE QUATERION PARA ANGULOS EULER #########

Roll: -4.01387e-05, Pitch: 0.000153481, Yaw: -0.000367328

######### CONVERSÃO DE ANGULOS EULER PARA MATRIZ DE ROTACAO #########

MATRIZ DE ROTACAO =
           1  4.00824e-05 -0.000367335
-4.01387e-05            1 -0.000153466
 0.000367328  0.000153481            1
######### CONSTRUCAO DA MATRIZ DE TRANSFORMACAO #########

MATRIZ DE TRANSFORMACAO =
           1  4.00824e-05 -0.000367335     0.480097
-4.01387e-05            1 -0.000153466  5.46884e-05
 0.000367328  0.000153481            1     0.679959
           0            0            0             1
```

Figura 4.28: Exemplo da mensagem recebida, das conversões e do cálculo da matriz de transformação geométrica ${}^{R}T_{H}$.

Por último, na subscrição do terceiro tópico, `/camera/rgb/aruco_tracker/transform`, obtém-se a transformação entre os sistemas de coordenadas do sensor e do *aruco marker*. A mensagem deste tópico também está em formato *quaterion*, sendo então necessário recorrer ao procedimento de conversões, explicado anteriormente, para que seja possível obter a transformação (4x4) respeitante à transformação ${}^{K}T_{A}$.

Verificou-se que a matriz de transformação (4x4) calculada pelo algoritmo não correspondia, em orientação, à transformação ${}^{K}T_{A}$ mas sim à transformação ${}^{K}T_{A_1}$ (figura 4.29). Neste trabalho, convencionou-se que a orientação do sistema de coordenadas do *aruco marker* era o da figura 4.21a.

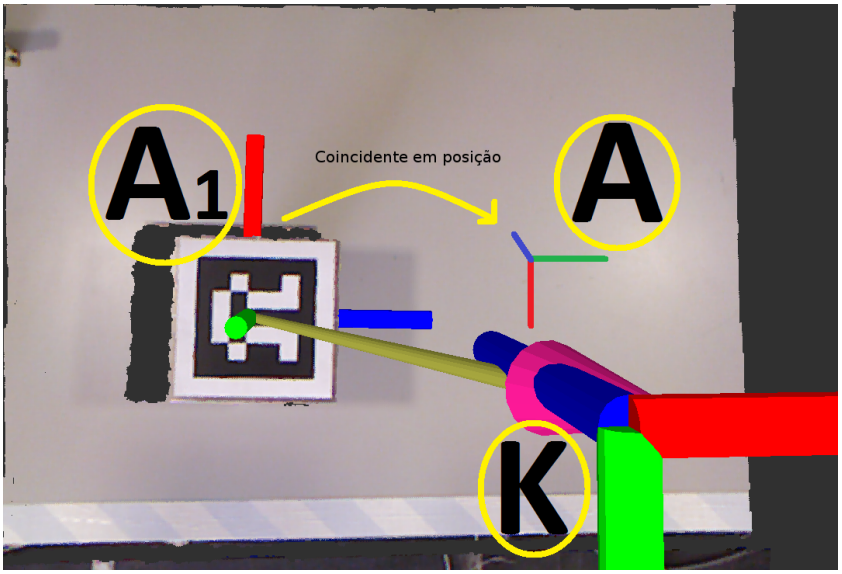


Figura 4.29: Representação dos sistemas de coordenadas do sensor e dos *aruco marker*.

Deste modo, calculou-se a transformação geométrica (4x4) ${}^{K}T_{A}$ que passasse pelo sistema de coordenadas A_1. Em termos de orientações, a matriz de transformação teórica que passa por A_1 é ${}^{K}T_{A}$ =Rot(z,$-\frac{\pi}{2}$)×Rot(x,$-\frac{\pi}{2}$)×Rot(y,π)×Rot(x,$-\frac{\pi}{2}$) origina a matriz de rotação teórica da seguinte transformação

$$T = \left[\begin{array}{ccc|c} 0 & 1 & 0 & p_x \\ 1 & 0 & 0 & p_y \\ 0 & 0 & -1 & p_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

A figura 4.30 representa a conversão e o cálculo da matriz de translação entre os dois sistemas de coordenadas. A matriz de transformação ${}^{K}T_{A}$ é válida, uma vez que a componente da rotação corresponde à matriz teórica, assim como a componente da posição.

De forma a aumentar a precisão dos valores obtidos pelo algoritmo do *aruco marker*, aproximou-se o mais possível o calibrador com o *aruco marker* do sensor. A figura 4.31

```
######### MENSAGEM RECEBIDA EM FORMATO QUATERION #########
[ INFO] [1468500891.242298121]: Sequence ID:[167322]
[ INFO] [1468500891.242361099]: msg time:[1468500891,239682513]
[ INFO] [1468500891.242417581]: tf[x=-0.006661,y=0.001162,z=0.253366]
[ INFO] [1468500891.242438662]: orientation[x=-0.503028,y=0.497847,z=-0.497016,w=0.502082]

######### CONVERSÃO DE QUATERION PARA ANGULOS EULER #########
Roll: -1.57257, Pitch: -0.000106321, Yaw: -1.56055

######### CONVERSÃO DE ANGULOS EULER PARA MATRIZ DE ROTACAO #########
MATRIZ DE ROTACAO =
  -0.0102462     0.999946   -0.0017771
    0.999947     0.010246 -0.000124536
-0.000106321  -0.00177829    -0.999998

MATRIZ DE TRANSFORMACAO =
  -0.0102462     0.999946   -0.0017771  -0.00666084
    0.999947     0.010246 -0.000124536   0.00116151
-0.000106321  -0.00177829    -0.999998     0.253366
           0            0            0            1
```

Figura 4.30: Exemplo da mensagem recebida, das conversões e do cálculo da matriz de transformação geométrica ${}^{K}T_{A}$.

representa quatro exemplos de quatro matrizes de transformação entre os sistemas de coordenadas do sensor e do *aruco marker*.

```
MATRIZ DE TRANSFORMACAO = (1)
  -0.0102462     0.999946   -0.0017771  -0.00666084
    0.999947     0.010246 -0.000124536   0.00116151
-0.000106321  -0.00177829    -0.999998     0.253366
           0            0            0            1

MATRIZ DE TRANSFORMACAO = (2)
 -0.0101518     0.999946 -0.00202996 -0.00666537
   0.999948    0.0101539  0.00105326  0.00118146
 0.00107382 -0.00201917    -0.999997     0.25337
          0            0            0           1

MATRIZ DE TRANSFORMACAO = (3)
 -0.0102006     0.999947  -0.0016671 -0.00665879
   0.999947    0.0102027  0.00123927   0.0011845
 0.00125622 -0.00165437    -0.999998    0.253352
          0            0            0           1

MATRIZ DE TRANSFORMACAO = (4)
 -0.0102051     0.999946 -0.00204363 -0.00666393
   0.999948    0.0102058 0.000330851  0.00116846
 0.00035169 -0.00204015    -0.999998    0.253369
          0            0            0           1
```

Figura 4.31: Quatro exemplares de matrizes de transformação geométrica ${}^{K}T_{A}$.

Os valores da transformação ${}^{R}T_{K}$ (resultado final da equação 4.2) estão presentes na figura 4.32. Os valores obtidos na matriz de transformação correspondem à transformação global entre os sistemas de coordenadas da base do robô e do sensor Kinect. Isto é, visto do sistemas de coordenadas da base do robô, a posição do sensor Kinect é de x=0,473 y=0,001 e z=0,933 m.

```
TransformacaoRoboToKinect=(1)
 -0.00975222     0.999955  -0.00423331     0.473309
    0.999932   0.00975635 -0.000676533   0.00119953
-0.000635299  -0.00423531    -0.999998     0.933759
           0            0            0            1

TransformacaoRoboToKinect= (2)
 -0.00971078     0.999941  -0.00493177     0.473293
    0.999953   0.00970683 -0.000823352   0.00119691
-0.000775432  -0.00493953    -0.999987     0.933743
           0            0            0            1

TransformacaoRoboToKinect= (3)
 -0.00975882     0.999943  -0.00423131     0.473308
    0.999952   0.00975605 -0.000676549   0.00119951
-0.000635229  -0.00423771    -0.999991     0.933758
           0            0            0            1

TransformacaoRoboToKinect= (4)
 -0.00968857     0.999945  -0.00413015     0.473307
    0.999953   0.00968688 -0.000427782   0.0012004
 -0.00038775  -0.0041341    -0.999991     0.933739
           0            0            0            1
```

Figura 4.32: Quatro exemplares de matrizes de transformação geométrica ${}^{R}T_{K}$.

De forma a analisar a qualidade dos dados obtidos, desenvolveu-se a análise das tabelas 4.1 e 4.2.

Os dados da tabela 4.1 correspondem às quatro matrizes da figura 4.31 enquanto que os da tabela 4.1 correspondem à figura 4.32. Cada célula da tabela corresponde à análise dos índices das matrizes. Por exemplo, o resultado a primeira célula diz respeito aos quatro valores da primeira linha e coluna de cada matriz. Esta análise foi realizada somente à componente de rotação da matriz de transformação geométrica, pois era a única que tinha valores teóricos. Não se obtiveram valores teóricos para a componente de translação, uma vez que era difícil obter-se um valor rigoroso.

Com a análise da tabela 4.1 verifica-se que o erro relativo máximo é de 1,020% e o erro relativo mínimo é de 0,002%. Com esta análise comprova-se que o método implementado é fiável e rigoroso, obtendo-se um erro baixo.

Tabela 4.1: Cálculo da média e do erro relativo dos valores de rotação das quatro matrizes ${}^{K}T_{A}$ da figura 4.31.

Média (Rad) = -0,010 Teórico (Rad) = 0 Erro (%) = 1,020	Média (Rad) = 0,999 Teórico (Rad) = 1 Erro (%) = 0,002	Média (Rad) = -0,009 Teórico (Rad) = 0 Erro (%) = 0,187
Média(Rad) = 0,999 Teórico (Rad) = 1 Erro (%) = 0,002	Média (Rad) = 0,010 Teórico (Rad) = 0 Erro (%) = 1,020	Média (Rad) = 0,001 Teórico (Rad) = 0 Erro (%) = 0,062
Média (Rad) = 0,001 Teórico (Rad) = 0 Erro (%) = 0,062	Média (Rad) = -0,002 Teórico (Rad) = 0 Erro (%) = 0,187	Média(Rad) = -0,999 Teórico (Rad) = -1 Erro (%) = 0,002

Com a análise da tabela 4.2 verifica-se que o erro relativo máximo é de 0,973% e o erro relativo mínimo é de 0,002%. Comparativamente com a tabela anterior, verifica-se que o erro máximo diminuiu, o que vem reforçar a precisão do método.

Tabela 4.2: Cálculo da média e do erro relativo dos valores de rotação das quatro matrizes ${}^{R}T_{K}$ da figura 4.32.

Média (Rad) = -0,009 Teórico (Rad) = 0 Erro (%) = 0,972	Média (Rad) = 0,999 Teórico (Rad) = 1 Erro (%) = 0,002	Média (Rad) = -0,004 Teórico (Rad) = 0 Erro (%) = 0,463
Média(Rad) = 0,999 Teórico (Rad) = 1 Erro (%) = 0,002	Média (Rad) = 0,009 Teórico (Rad) = 0 Erro (%) = 0,972	Média (Rad) = -0,001 Teórico (Rad) = 0 Erro (%) = 0,065
Média (Rad) = -0,001 Teórico (Rad) = 0 Erro (%) = 0,060	Média (Rad) = -0,004 Teórico (Rad) = 0 Erro (%) = 0,438	Média(Rad) = -0,999 Teórico (Rad) = -1 Erro (%) = 0,002

De forma a testar a validade de toda a metodologia, alterou-se a posição de calibração do robô e analisaram-se as novas matrizes de transformação da base do manipulador para o sensor. Como seria de esperar, os valores de posição continuaram iguais, embora fosse

notável que quanto mais o *aruco marker* estivesse afastado da posição do sensor maior era a variação dos valores da matriz final.

Com esta verificação, garantiu-se que a metodologia de calibração automática foi bem desenvolvida, aliando a rapidez na obtenção da matriz com a fiabilidade de todos os valores obtidos.

Capítulo 5

Perceção visual

De forma a validar a aplicabilidade, a robustez e versatilidade da API, desenvolveu-se uma aplicação demonstrativa que consistiu no jogo de damas contemplando a cooperação entre o manipulador e o utilizador. Para se comandar o robô com base em tomada de decisões, identificou-se a posição do tabuleiro e a disposição das peças. Uma vez que o material utilizado até ao momento não permitia solucionar tal problema, recorreu-se ao sensor Kinect. A figura 5.1 enquadra o sensor Kinect que recolhe e envia a informação para o computador.

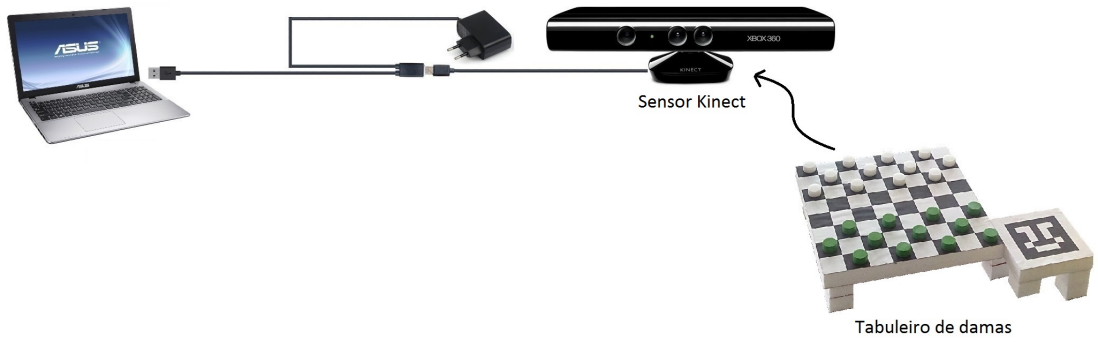


Figura 5.1: Esquema de utilização do hardware auxiliar à visão artificial.

Nesta lógica, a descrição deste capítulo centrar-se-á nos processos que foram criados para tratar a informação proveniente do sensor. A informação do sensor pode dividir-se em duas grandes categorias, a informação de nuvens de pontos 3D e a informação de imagem 2D. Neste trabalho, a primeira metodologia adotada foi a análise das nuvens de pontos, descrita na secção 5.1. Por falta de precisão, esta metodologia verificou-se como insuficiente para a aplicação pretendida, optando-se pela informação do sensor RGB, descrita na secção 5.2.

5.1 Metodologia por nuvens de pontos

Esta secção descreve a metodologia das nuvens de pontos captadas pelo sensor Kinect. Para esta metodologia, utilizou-se a biblioteca PCL. Tendo como referência o sensor Kinect, as nuvens de pontos são captadas num intervalo de 800 mm e 3500 mm. Segundo [39], uma nuvem de pontos tem a dimensão máxima de 640x480.

A distância entre o sensor e a mesa de trabalho foi de 890 mm. Visto que as peças têm uma dimensão de 17 mm e se encontram na superfície do tabuleiro, que se localiza a 130 mm do topo da mesa de trabalho, considera-se que a distância teórica entre o sensor e as superfícies das peças é 743 mm. A figura 5.2 apresenta as medidas entre cada componente.

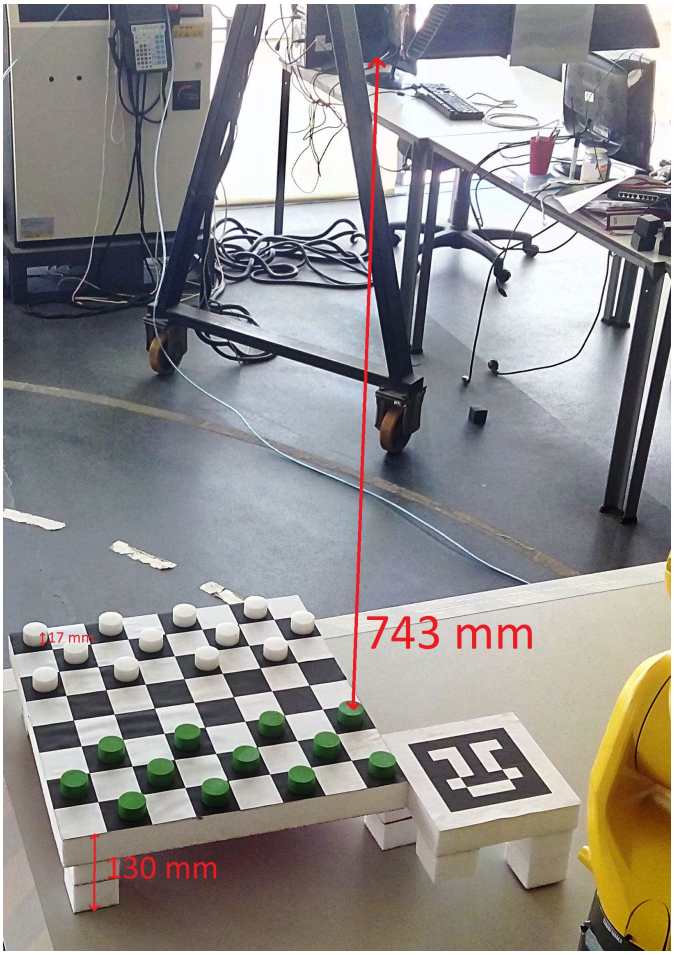


Figura 5.2: Distâncias da superfície das peças para o sensor e alturas do tabuleiro e das peças.

Criou-se uma sequência que processasse a informação recebida do sensor Kinect e determinasse as coordenadas de posição que o robô teria de ocupar, respeitando as regras do jogo. Para tal, foi concebida a sequência da figura 5.3.

Em ambiente ROS, a sequência demonstrada anteriormente apresenta-se sob a forma da figura 5.4. A transformação Kinect/Aruco e a nuvem de pontos provêm da arquitetura

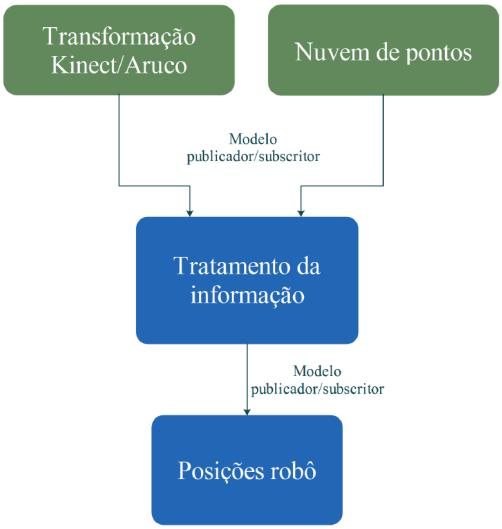


Figura 5.3: Relação entre os tópicos (a verde) e os nodos do método por nuvens de pontos (a azul).

do sensor Kinect (a verde). Por outro lado, o tratamento da informação é composto por 7 nodos e representa o processo de tratamento da nuvem de pontos desde a recolha por parte do sensor até à identificação das peças (a azul).

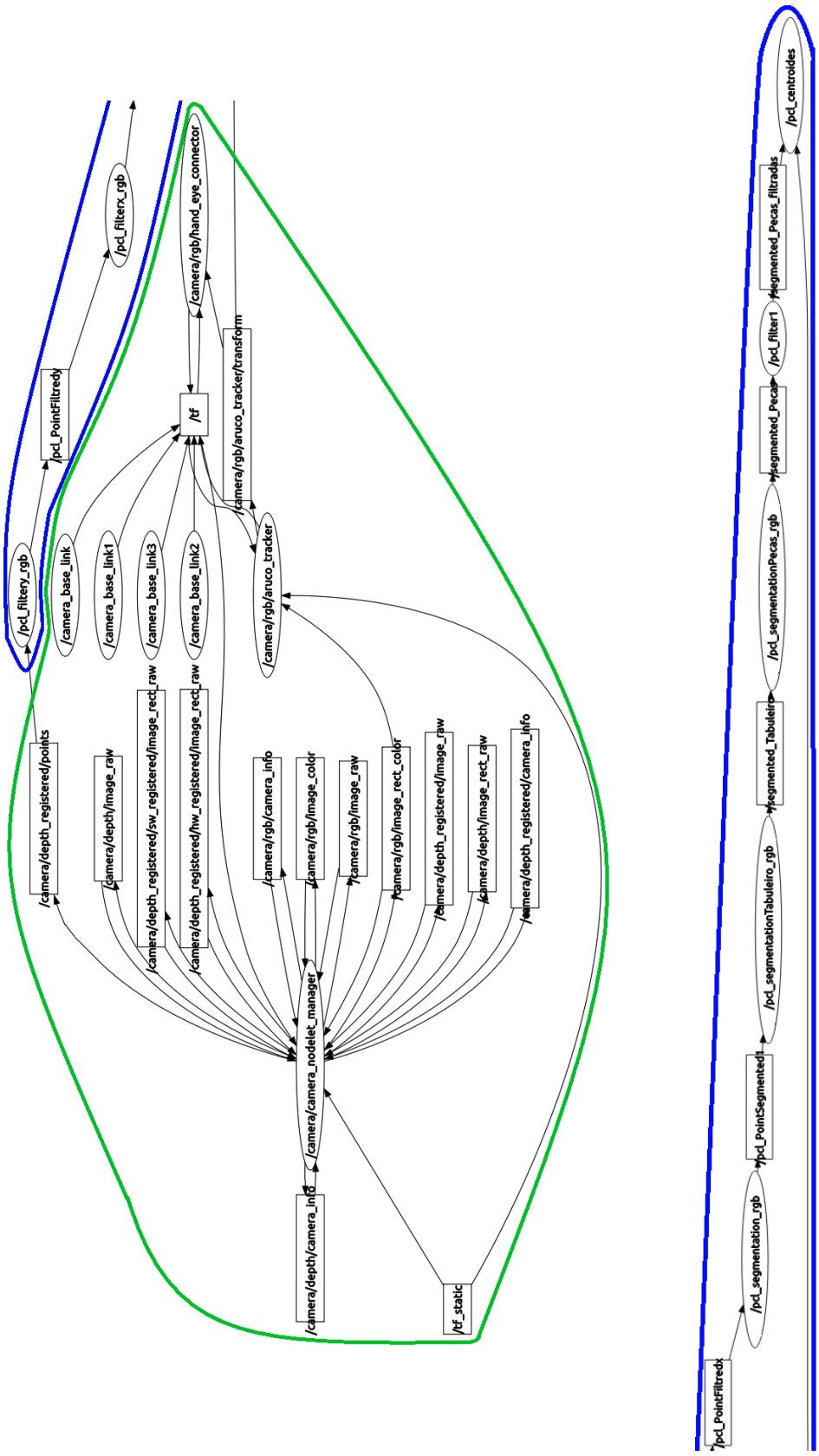
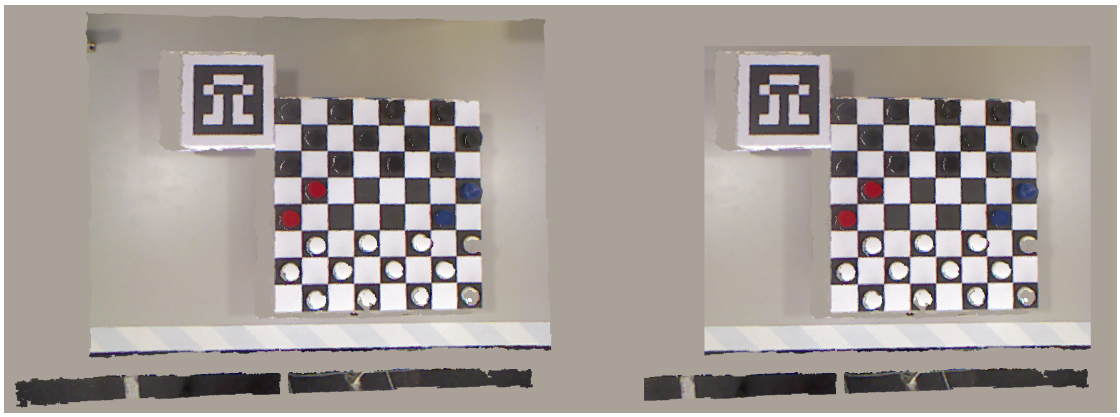


Figura 5.4: Arquitetura geral ROS da metodologia por nuvens de pontos.

Quando os nodos ROS subscrevem uma mensagem, necessitam de converter a mensagem da nuvem para o formato PCL. O oposto também acontece, quando se pretende publicar uma determinada nuvem de pontos num tópico. Utiliza-se a função `pcl::fromROSMsg` e `pcl::toROSMsg` para cada um dos casos mencionados. Esta conversão ocorre nos 7 nodos criados.

`Tratamento da informação`

O primeiro nodo ROS (`pcl_filtery_rgb`) subscreve o tópico que contém a mensagem da nuvem de pontos recolhida pelo sensor (`/camera/depth_registered/points`). Este nodo cria uma segunda nuvem, que sofreu uma filtragem. O filtro aplicado limita a coordenada y da nova nuvem de pontos, eliminando da nuvem as regiões que o manipulador não atingia. Posteriormente, a nova nuvem de pontos foi publicada no tópico `pcl_PointFiltredy`. O nodo `pcl_filterx_rgb` tem exatamente o mesmo objetivo que o nodo anterior, diferenciando-se pelo facto do limite ocorrer na coordenada x. As figuras 5.5a e 5.5b representam a diferença das nuvens de pontos.



(a) Nuvem de pontos original. (b) Nuvem de pontos filtrada em x e y.

Figura 5.5: Representação da nuvem de pontos onde se verifica a influência das limitações.

O terceiro nodo ROS, `pcl_planar_segmentation_rgb`, aplica a primeira segmentação à nuvem de pontos. Aplicou-se o *Random Sample Consensus* (RANSAC), um método iterativo para estimar os pontos pertencentes a um modelo. O modelo utilizado foi um plano, definindo-se um número máximo de iterações de 1000. Após o algoritmo ter definido um plano, consideraram-se válidos todos os pontos com coordenadas entre o plano e o sensor. Como seria de esperar, o plano considerado pelo algoritmo foi o plano da mesa de trabalho e os pontos removidos foram todos aqueles além da mesa (figura 5.6).

Sendo o principal objetivo isolar as superfície das peças, criou-se uma solução que dependesse do tabuleiro, pois a posição ou uma ligeira inclinação deste poderia comprometer a segmentação. Desta forma, criou-se o nodo ROS `pcl_planar_segmentationTabuleiro_rgb`. Este utiliza a mesma estratégia de obtenção do plano que o nodo anterior. Tendo o plano da superfície do tabuleiro selecionado, definiu-se um novo plano imaginário distanciado 10 mm do plano da mesa. Considerou-se que os pontos eram tidos como válidos se estivessem entre o plano imaginário e o sensor.

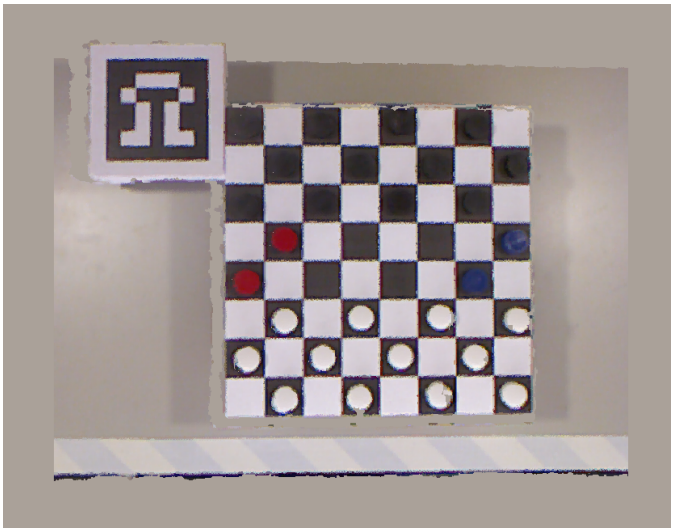
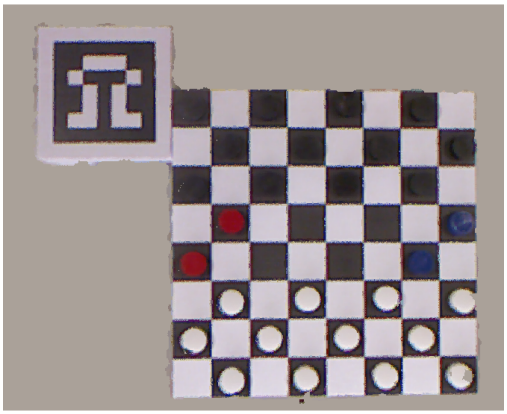
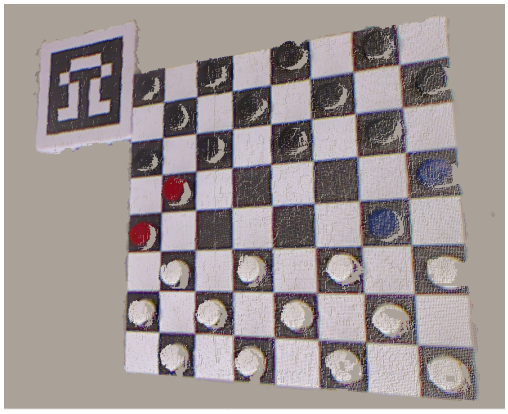


Figura 5.6: Nuvem de pontos até ao plano da mesa e da região de trabalho do robô.

Com isto, garante-se que a mesa e todos os pontos 10 mm acima do plano da mesa são removidos. Esta metodologia garante uma independência da posição e das orientações que o tabuleiro possa tomar (figuras 5.7a e 5.7b).



(a) Nuvem de pontos sem inclinação.



(b) Nuvem de pontos com uma ligeira inclinação.

Figura 5.7: Nuvem de pontos da superfície do tabuleiro e das peças, com e sem inclinação do tabuleiro.

O nodo ROS `pcl_planar_segmentationPecas_rgb` corresponde ao nodo que extrai as superfícies de todas as peças presentes no topo do tabuleiro. Utiliza o mesmo algoritmo e o mesmo procedimento do plano imaginário do nodo anterior. Como a superfície da nuvem de pontos tem associada um nível de ruído, o valor da distância teve de ser escolhido com base num processo iterativo. Deste modo, e como as peças têm uma altura de 17 mm, definiu-se que o plano imaginário acima do plano do tabuleiro estaria distanciado 10 mm. Assim preservam-se as superfícies das peças (figura 5.8).

O nodo ROS `pcl_filter1` aplica o filtro para remover os *outliers* da nuvem de

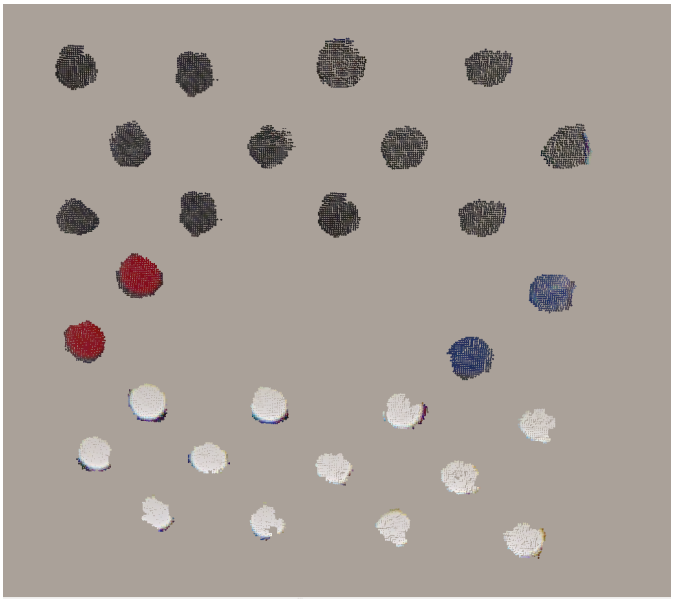


Figura 5.8: Nuvem de pontos correspondente às superfícies de todas as peças presentes no tabuleiro.

pontos das superfícies das peças. Este filtro baseia-se no raio de vizinhança de todos os pontos presentes na nuvem. Definiu-se que cada ponto, num raio de 10 mm, teria de ter um mínimo de 40 pontos vizinhos. Garantiu-se não só a presença da informação das superfícies das peças como a eliminação de qualquer ruído pontual de pequena dimensão.

Por último, e não menos importante, o nodo ROS `pcl_centroides`. Neste nodo, como se verifica na figura 5.4, é onde ocorre a subscrição dos dois tópicos, `/segmented_Pecas_filtradas` e `/camera/rgb/aruco_traker/transform`. O primeiro tópico contém a informação da nuvem de pontos tratada, a qual foi explicada até ao momento. Por outro lado, o segundo tópico comporta a mensagem da transformação geométrica instantânea entre o sensor Kinect e o *aruco marker*. A estrutura de cálculo da matriz de transformação ${}^{K}T_{A}$ é igual à criada na figura 4.30.

A mensagem com a nuvem de pontos disputa a concretização dos principais objetivos deste nodo, que são:

- separar as cores da nuvem de pontos recebida;
- identificar e isolar cada um dos aglomerados de cada nuvem de pontos;
- calcular os seus centróides;
- converter as coordenadas;
- construir o tabuleiro virtual;
- enviar a disposição do tabuleiro virtual para nodo que possui o algoritmo de jogo;

A separação das quatro cores, correspondente às quatro cores de jogo, é feita por HSV. Como o formato original da nuvem de pontos é em RGB, recorreu-se à função da PCL

`PointCloudXYZRGBtoXYZHSV` para converter a nuvem de RGB em HSV. Posteriormente, isolaram-se as quatro cores em quatro nuvens para que as diferentes peças pudessem ser analisadas.

Para cada uma das quatro nuvens de pontos, isolou-se cada aglomerado de pontos. Cada um dos aglomerados corresponde a uma superfície de uma peça no tabuleiro. Para tal, recorreu-se à função da PCL `EuclideanClusterExtraction`. Esta função isola todos os aglomerados de pontos presentes numa nuvem de pontos.

Para confirmar que a separação das cores e dos aglomerados tinha sido realizada com sucesso, implementou-se um contador em cada uma das análises das nuvens. Por exemplo, na nuvem da figura 5.8, verificou-se que as cores e os aglomerados foram bem identificados. Concluiu-se que a metodologia adotada era válida e que os parâmetros considerados estavam devidamente ajustados (figura 5.9).

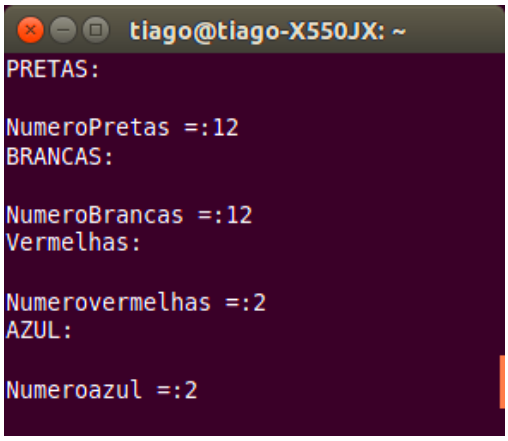


Figura 5.9: Contagem e diferenciação de cada aglomerado da nuvem de pontos, correspondente à figura 5.8.

Com a informação de cada cor separada e devidamente identificada, procedeu-se ao cálculo de cada centróide e à conversão das coordenadas em relação ao sistema de coordenadas do *aruco marker*. Para obter o cálculo do centróide, recorreu-se à função da PCL `compute3DCentroid`, a qual recebe como parâmetro de entrada cada índice calculado anteriormente e devolve as coordenadas do centróide do índice em causa. Uma vez que as coordenadas dos centróides são calculadas em relação ao sistema de coordenadas do sensor e se pretende construir um tabuleiro virtual, converteram-se as coordenadas do sistema de coordenadas do sensor para o sistema de coordenadas do *aruco marker*.

Com recurso à primeira equação da equação 4.3, calculou-se a transformação geométrica ${}^{A}T_{P}$ que corresponde à conversão das coordenadas de cada centróide vistas do sistema de coordenadas do *aruco marker*. Na figura 5.10 está presente a nuvem das superfícies das peças e o posicionamento dos sistemas de coordenadas.

O procedimento seguinte foi a construção do tabuleiro virtual, que contém a disposição instantânea do tabuleiro. A finalidade do tabuleiro virtual é o envio da disposição das peças no tabuleiro para o algoritmo de jogo. Foi na construção do tabuleiro que se verificou que a metodologia por nuvens de pontos não era a metodologia que garantia fiabilidade e precisão da informação recolhida.

O primeiro problema a surgir foi a qualidade dos dados recebidos. A qualidade da

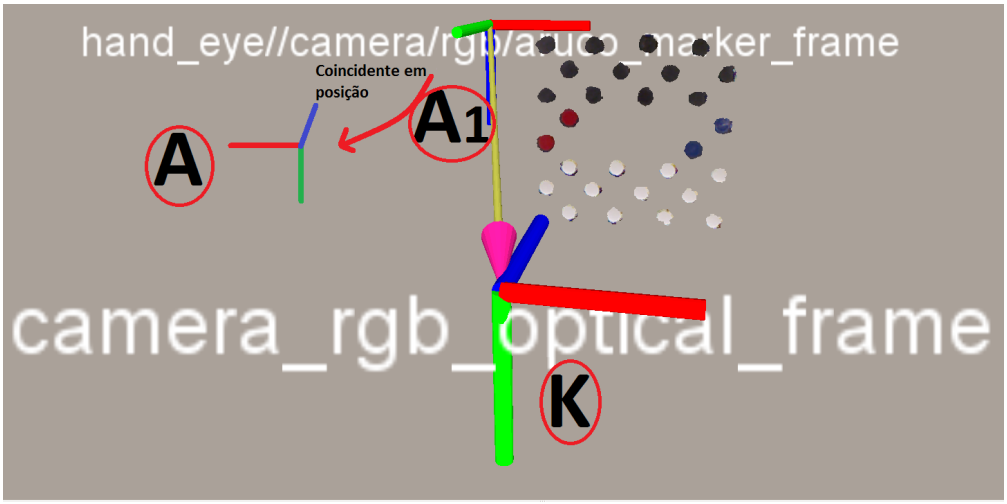


Figura 5.10: Nuvem de pontos das superfícies das peças com a representação das posição e orientação dos sistemas de coordenadas do *aruco marker* e do sensor Kinect.

informação obtida compromete não só a construção do tabuleiro mas também o cálculo das jogadas. Como se verifica nas figuras 5.11a, 5.11b, 5.11c e 5.11d, a quantidade de pontos em cada aglomerado varia, deixando, por vezes, de ser identificada. De notar que este défice de informação está associado a fatores como a reflexão da luz, a distância entre o sensor e as peças, o uso de peças de pequenas dimensões e a resolução do sensor. As peças com maior índice de absorção têm uma definição mais constante, enquanto que peças com maior índice de reflexão, por exemplo as peças brancas, a qualidade da informação nem sempre é a melhor.

O segundo problema surgiu na atribuição das posições do tabuleiro virtual a cada aglomerado. Uma vez que as peças têm dimensões reduzidas, verificou-se que para as peças com melhor informação, o valor do centróide tinha uma variação de ± 10 mm. A necessidade de uma elevada precisão dos centróides surgiu para a construção do tabuleiro virtual. Por outro lado, a ventosa presente no *end-effector* tem um diâmetro de 15 mm e as peças tem um diâmetro de 30 mm, ficando com 15 mm de sobra. Desta forma temos uma margem de erro de 7.5 mm para cada lado, necessitando de precisões elevadas. Com a baixa precisão dos valores dos centróides, a construção do tabuleiro torna-se uma tarefa inexequível.

Outra conclusão a que se chegou foi que o tratamento da nuvem de pontos, desde a captação até à segmentação das peças, era um processo muito exigente computacionalmente. Este facto fazia com que uma jogada demorasse 2 minutos a ser calculada.

Com estas impossibilidades, suspendeu-se o desenvolvimento desta metodologia e optou-se pelo outro recurso do sensor Kinect, a imagem rgb.

(a) Superfícies das peças incompletas.

(b) Superfícies das peças incompletas.

(c) Défice de peças.

(d) Imperfeição da informação recolhida.

Figura 5.11: Nuvens de pontos das superfícies das peças com défice de qualidade.

5.2 Metodologia por imagens 2D

Esta secção refere-se à metodologia de imagens captadas pelo sensor Kinect. Criou-se e desenvolveu-se uma arquitetura constituída por três nodos ROS que comunicam entre si de diferentes modos (figura 5.12). A figura 5.12 pretende transmitir a ideia geral do funcionamento da metodologia desenvolvida que inicia com o nodo principal a subscrever dois tópicos e a processar a informação proveniente dos mesmos. Posteriormente, envia um tabuleiro virtual com a disposição das peças que se encontram no tabuleiro para o nodo algoritmo de jogo e aguarda que as jogadas sejam calculadas. Após a receção da jogada, o nodo principal conclui o seu processo e envia uma mensagem para o nodo coordenadas de posição do robô.

Para esta metodologia, utilizou-se a biblioteca OpenCV. A explicação do `nodo principal` ocorre na presente secção, quanto que a dos nodos `Algoritmo de jogo` e `Coordenadas de posição do robô` ocorrem nas secções 6.1 e 6.3, respetivamente.

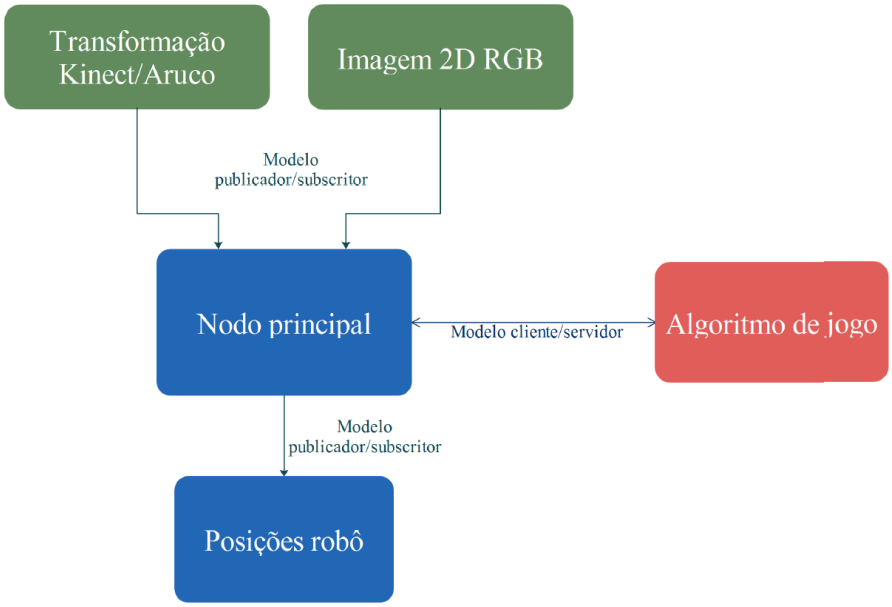


Figura 5.12: Relação entre os tópicos (a verde), os nodos principais da metodologia por imagens 2D (a azul) e os nodos auxiliares (a vermelho).

Quando são recebidas as mensagens dos tópicos da `imagem 2D RGB` e da `transformação Kinect/Aruco` é executada a `função posição aruco` e a `função tratamento`, respetivamente. Cada uma das funções tem uma sequência de tarefas bem definidas (figura 5.13).

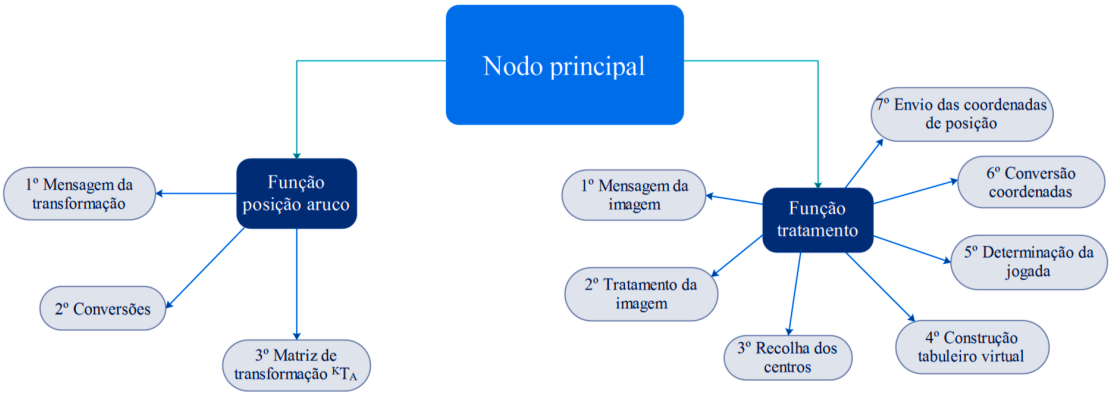


Figura 5.13: Funções criadas no interior do nodo principal.

5.2.1 `Função posição aruco`

A `função posição aruco` é executada quando o `nodo principal` recebe uma nova mensagem da transformação entre os dois sistemas de coordenadas. A estrutura de cálculo da matriz de transformação $^{K}T_{A}$ é similar à criada na figura 4.30. A matriz de transformação geométrica fica guardada numa variável global.

5.2.2 `Função tratamento`

A `função tratamento` é executada quando é recebida a mensagem de uma imagem RGB. Engloba a principal sequência de tarefas que culminaram no envio das coordenadas de posição que o robô terá de ocupar. Como a sequência de tarefas desta função é extensa, dividiu-se a sua explicação em dois esquemas, sendo o primeiro esquema (figura 5.14) o estudado nesta secção.

Conforme a informação da figura 5.14, quando a `função tratamento` recebe uma imagem nova, realizam-se duas conversões e criam-se novas imagens baseadas nas cores da imagem original. Para cada uma das novas imagens são realizadas operações de tratamento de imagem. Após se identificar a informação de cada imagem, avança-se para um processo de decisão de forma a prosseguir com a análise da imagem. Para imagens em que se verifique a presença de pixels, calculam-se os centróides de cada aglomerado de pixels e verifica-se se a posição de cada aglomerado corresponde a uma posição válida do tabuleiro de jogo.

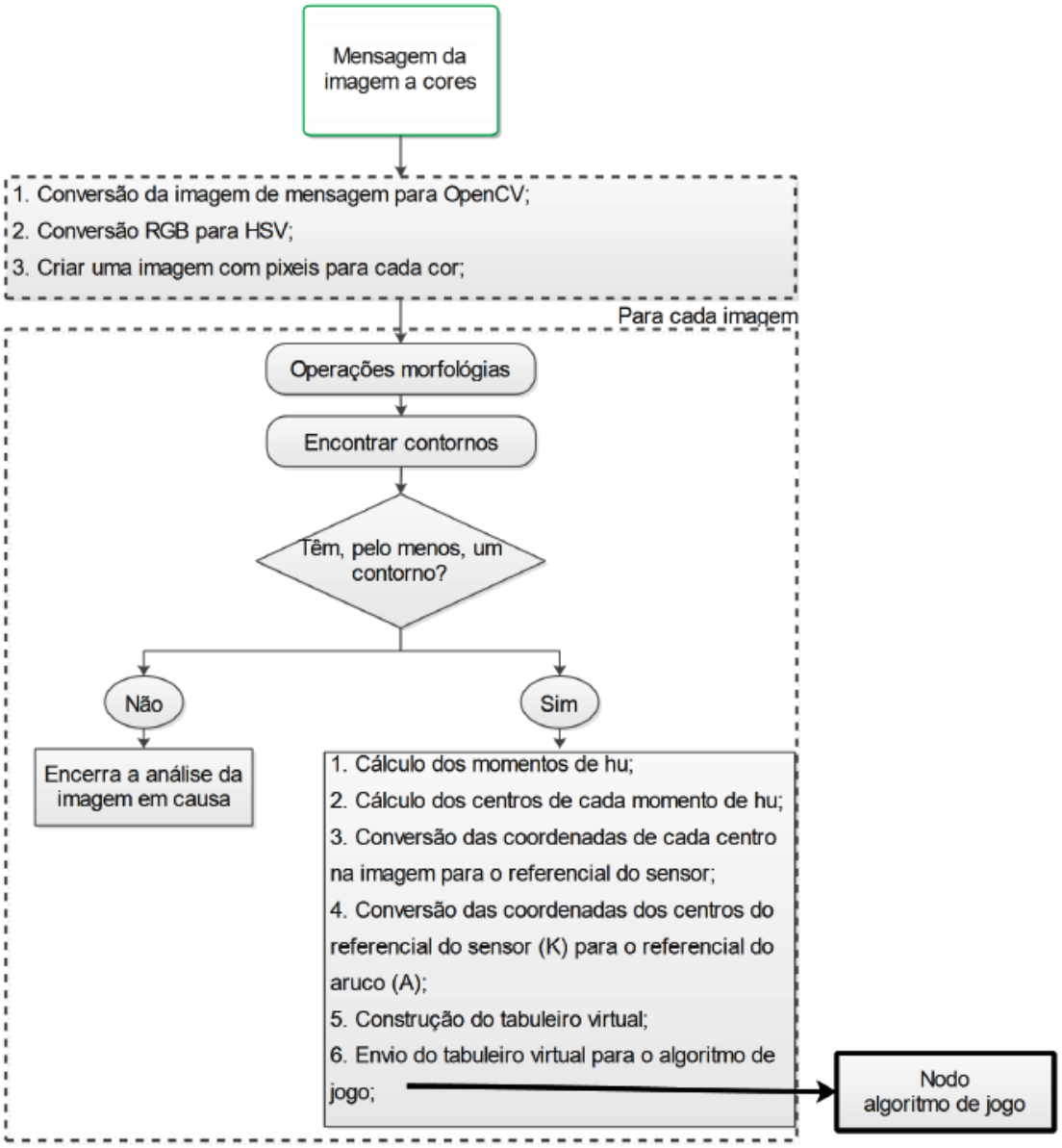
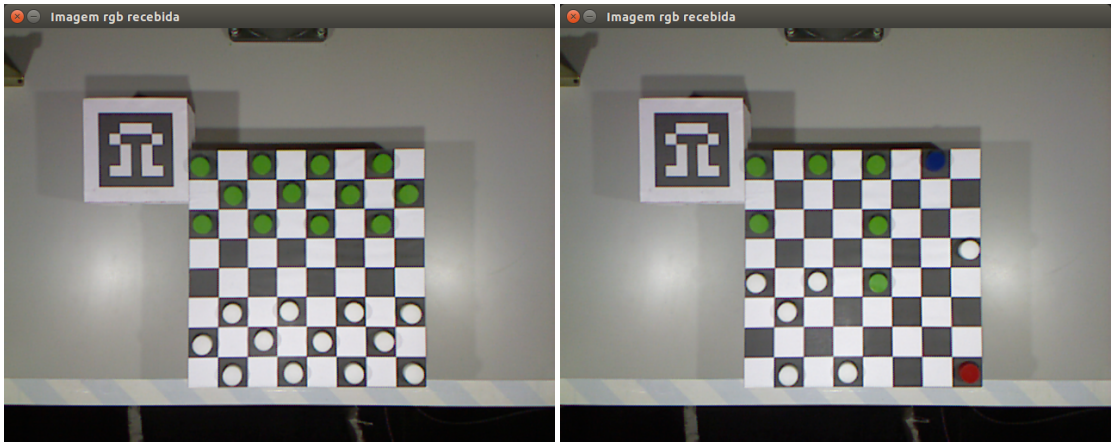


Figura 5.14: Detalhe da estrutura da função tratamento até ao algoritmo de jogo.

Após a receção da mensagem da imagem, proveniente do tópico `/camera/rgb/image_color`, converteu-se o seu formato de mensagem para OpenCV tornando-se possível utilizar a biblioteca OpenCV. A imagem é recebida em formato RGB. As figuras 5.15a e 5.15b, correspondem a duas imagens recebidas em duas situações de jogo distintas, à situação de início de jogo e à situação de um jogo corrente, respetivamente.



(a) Disposição inicial do jogo. (b) Disposição num momento de jogo.

Figura 5.15: Imagens originais captadas em duas situações distintas.

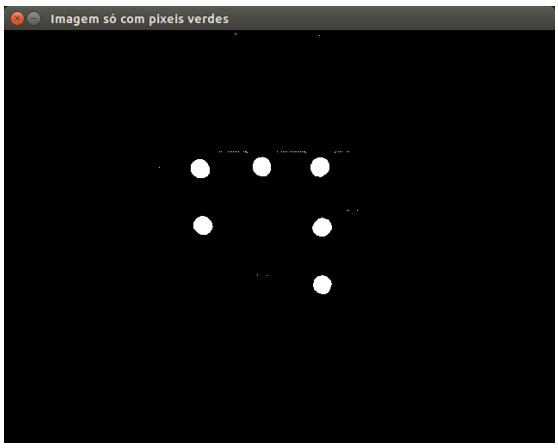
Para diferenciar as peças que correspondem a cada jogador, separam-se as cores das peças. Para separar as cores de forma mais precisa, converteram-se as cores de RGB em HSV. Recorrendo à função `inRange` da biblioteca OpenCV, procedeu-se à separação das cores das peças, especificando-se o intervalo de valores para o HSV. Por exemplo, para a cor vermelha, especificou-se que os valores de H teriam de estar entre 10 e 350, quanto que os de S e V eram 0.85 a 1 e 0.90 a 1, respetivamente.

Pelo método das nuvens de pontos, era possível distinguir a presença de peças pretas presentes em quadrados pretos. Por sua vez, na imagem 2D esta distinção tornou-me mais difícil. Por isso, e de forma a minimizar a ocorrência de ambiguidades e de erros, coloriu-se as peças pretas de verde. Assim sendo, as cores do jogo passaram a ser quatro, verde e branca para os peões e azuis e vermelhas para as damas.

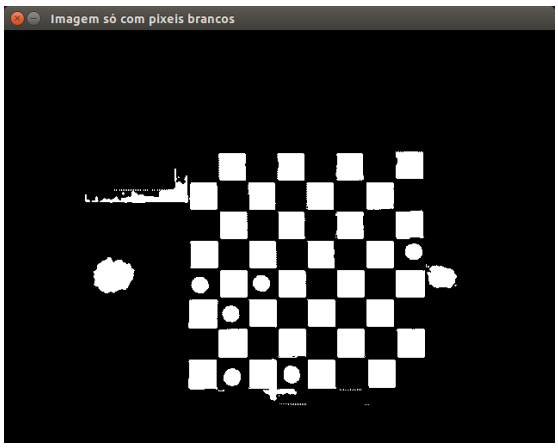
A figura 5.15b serve de referência para a explicação que se segue.

Com o processo de selecionar o intervalo de valores de HSV, obtiveram-se as imagens das figuras 5.16a, 5.16b, 5.16c e 5.16d correspondentes ao intervalo de valores das cores dos pixels para os verdes, brancos, vermelhos e azuis. É notória a presença de informação adicional nessas imagens, isto é, informação que não corresponde a posições válidas para o jogo.

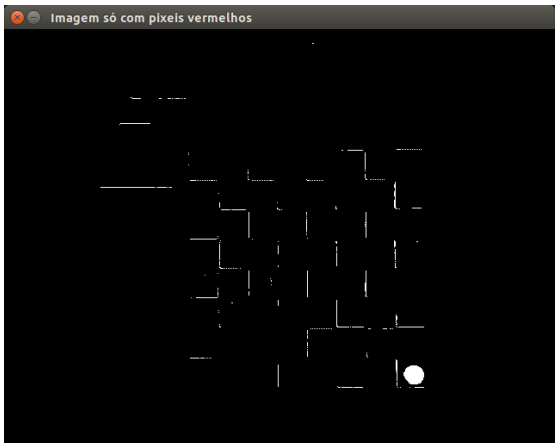
De forma a eliminar ruído das imagens, recorreu-se a operações morfológicas. Como o ruído era essencialmente composto por pontos isolados e aglomerados de pontos com uma área inferior à área de uma peça, realizaram-se duas operações de morfologia, um fecho e uma abertura. Por definição, um fecho define-se como uma dilatação seguida de uma erosão . Por outro lado, a abertura define-se como uma erosão seguida de uma dilatação. Desta forma, a qualidade da informação recolhida melhorou (figuras 5.17a, 5.17b, 5.17c e 5.17d), sendo possível calcular os centros de cada aglomerado de pixels com maior



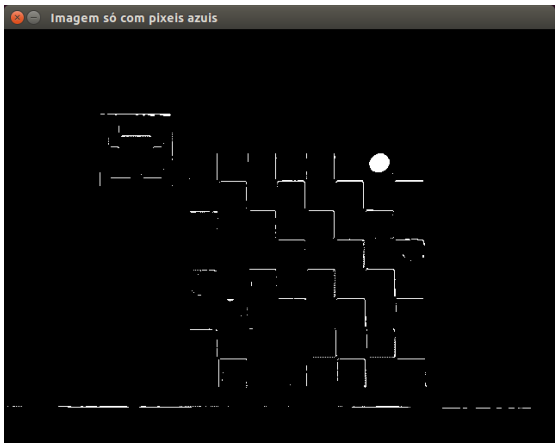
(a) Imagem referente aos pixels verdes.



(b) Imagem referente aos pixels brancos.



(c) Imagem referente aos pixels vermelhos.

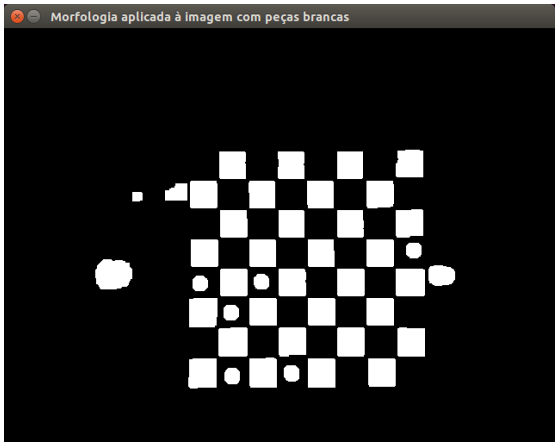


(d) Imagem referente aos pixels azuis.

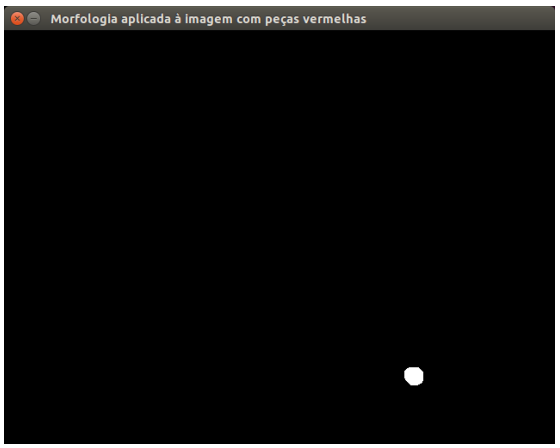
Figura 5.16: Separação dos pixels correspondentes a cada uma das cores da imagem.

precisão. O elemento estruturante para as duas operações morfológicas anteriores foi uma matriz quadrada de uns, com dimensão 9x9.

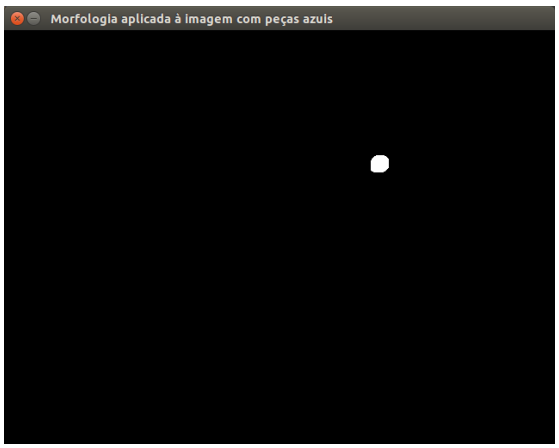
(a) Imagem referentes aos pixels verdes sem ruído.



(b) Imagem referentes aos pixels brancos com algum ruído.



(c) Imagem referentes aos pixels vermelhos sem ruído.



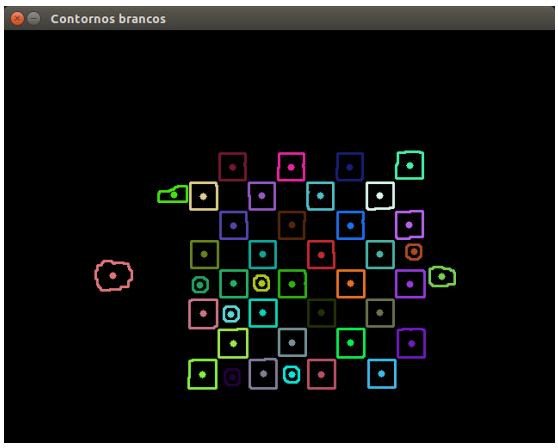
(d) Imagem referentes aos pixels azuis sem ruído .

Figura 5.17: Tratamento da imagem de forma a isolar as regiões de interesse.

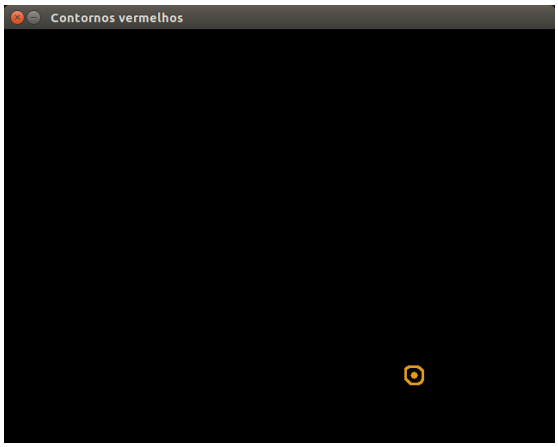
O passo seguinte foi isolar os aglomerados de pixels, definindo um limite em torno de cada um dos aglomerados (figuras 5.18a, 5.18b, 5.18c e 5.18d). A função *findContours* foi a utilizada para definir os contornos. A função retorna o número total de contornos identificados numa imagem. O número total de contornos é utilizado como fator de decisão. Nos casos em que não se verifiquem contornos numa imagem, a analise da imagem em causa é encerrada, poupando tempo computacional. Se o valor estiver compreendido entre 1 e 12 significa que foram detetadas peças.

Uma vez que a imagem 5.18b contém bastante ruído, como por exemplo o facto dos quadrados do tabuleiro também serem brancos, esta etapa não é determinante para esta imagem. A solução adotada para diferenciar se um contorno correspondia a uma peça ou a ruído foi implementada na construção do tabuleiro virtual. Na construção do tabuleiro virtual criou-se uma metodologia que, através dos centróides das peças, se verificava a validade das posições.

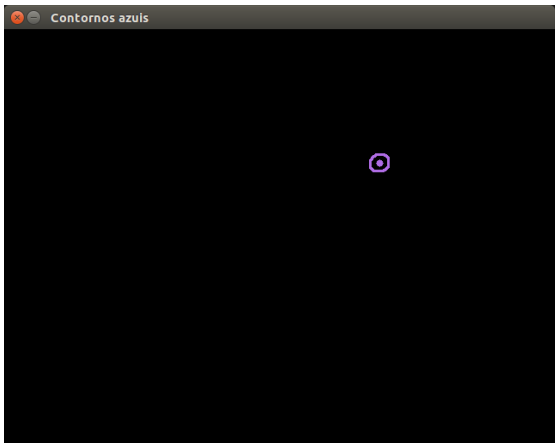
(a) Imagem com contornos dos pixels verdes.



(b) Imagem com contornos dos pixels brancos.



(c) Imagem com contornos dos pixels vermelhos.



(d) Imagem com contornos dos pixels azuis.

Figura 5.18: Contornos das regiões onde será realizado o cálculo do centróide.

A explicação seguinte é focada numa imagem que contém, pelo menos, um contorno. Com a informação do contorno de cada aglomerado de uma imagem, calcularam-se as suas posições em relação ao sistema de coordenadas da imagem, que tem a sua origem no canto superior esquerdo. Utilizaram-se os momentos invariantes, que permitem o cálculo da área de um objeto bem como o seu centróide. Esse cálculo foi realizado para todos os aglomerados de pixels de cada imagem, utilizando as duas equações para o cálculo dos centróides de cada um.

$\bar{x}= \frac{M_{10}}{M_{00}}$

$\bar{y}= \frac{M_{01}}{M_{00}}$

O eixo da abcissa do sistema de coordenadas da imagem é coincidente com a maior dimensão da imagem (640 pixels) e o eixo das ordenadas é coincidente com a menor dimensão da imagem (480 pixels).

O próximo objetivo passava por transformar as coordenadas de cada um dos centróides do sistema de coordenadas da imagem para o sistema de coordenadas do sensor. O sistema de coordenadas do sensor tem origem no centro da imagem e tem a mesma orientação que o sistema de coordenadas da imagem. Esta transformação consegue-se à custa dos parâmetros intrínsecos da câmara e com os pixels da imagem, isto é, conversão de pixels para coordenadas do mundo. Para realizar esta transformação, utilizaram-se as equações 5.1 e 5.2, em que os termos mc_x e mc_y correspondem aos centróides em pixels, C_x e C_y ao centro da imagem, z_{world} a distância das peças ao sensor e, por último, o f_x e f_y correspondente à distância focal. Os valores da distância focal e o centro da imagem são retirados diretamente da matriz K dos parâmetros intrínsecos da câmara.

$$x_{world} = \frac{(mc_x - C_x) \times z_{world}}{f_x} \tag{5.1}$$

$$y_{world} = \frac{(mc_y - C_y) \times z_{world}}{f_y} \tag{5.2}$$

De forma a definir uma origem que dependa da posição do tabuleiro, transformaram-se as coordenadas dos centróides do sistema de coordenadas do sensor Kinect para o do *aruco marker*. No fundo, a transformação geométrica pretendida é ${}^{A}T_{P}$, da equação 4.3. Como já se calcularam as coordenadas das peças em relação ao sistema de coordenadas do sensor Kinect, ${}^{K}T_{P}$, e se tem a transformação entre o sensor e o *aruco marker*, ${}^{K}T_{A}$, aplica-se a equação 4.3 e obtêm-se as coordenadas em relação ao sistema de coordenadas do *aruco marker*.

Com isto, estamos em condições de criar um tabuleiro virtual. Convencionou-se que o tabuleiro virtual era numerado de 0 a 4, sendo o 0 uma posição sem peça, o 1 um peão branco, 2 um peão verde, o 3 uma dama azul e por último, o 4, correspondendo a uma dama vermelha. Por defeito o tabuleiro é constituído por 0 e a sua atualização é realizada por meio de uma função que se intitulava `ConversaoTabuleiro`.

5.2.2.1 `Função ConversaoTabuleiro`

Esta função tem início com a conversão de coordenadas métricas para posições de tabuleiro. O principio de construção do tabuleiro virtual desenvolvido apresenta-se na figura 5.19. A mesma figura contém três constantes d, x_{off} e y_{off}. O valor de d é 0.050 m, correspondendo à dimensão real de cada quadrado do tabuleiro. Por outro lado, e como não foi possível coincidir o centro do sistema de coordenadas do *aruco marker* com o canto superior esquerdo do tabuleiro real, criou-se um *offset*. O valor do *offset* em x (x_{off}) é de 0.127 m enquanto que o valor do *offset* em y (y_{off}) é de 0.0325 m.

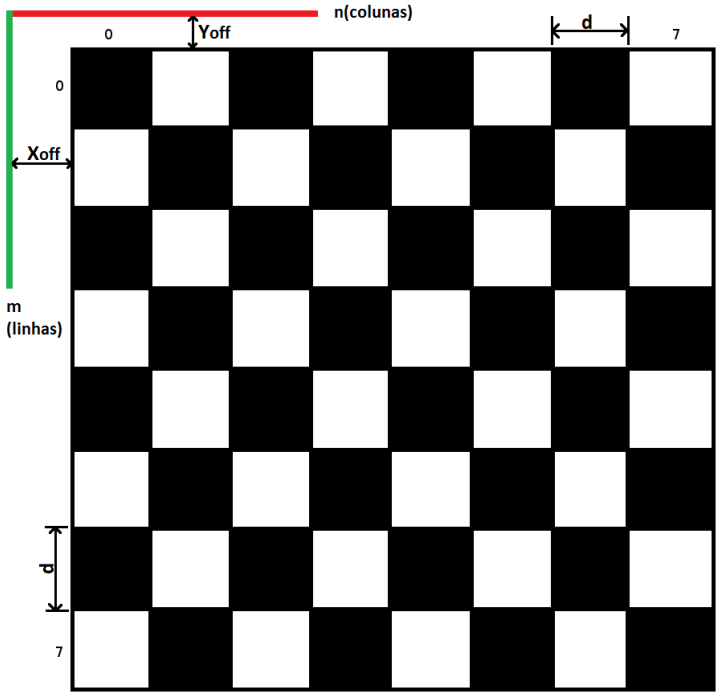


Figura 5.19: Tabuleiro virtual.

Desenvolveu-se, de igual modo, um conjunto de equações que proporcionaram a conversão de coordenadas métricas para posições de tabuleiro e vice-versa. As equações 5.3 e 5.4 convertem posições do tabuleiro em coordenadas métricas, enquanto que as equações 5.5 e 5.6 fazem o processo inverso.

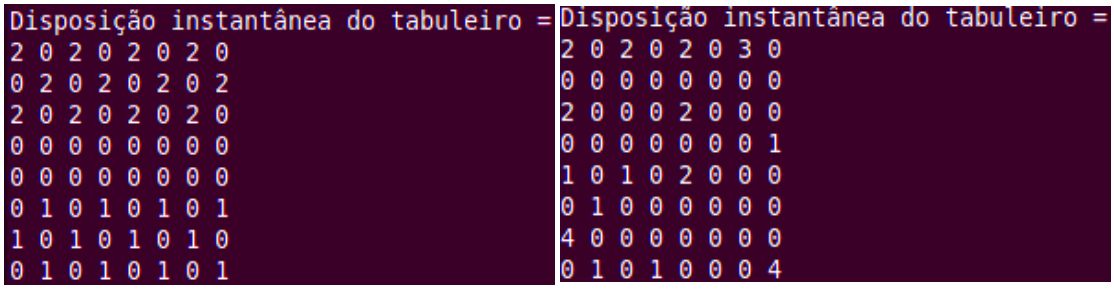
$$x = x_{off} + n \times d + \frac{d}{2} \tag{5.3}$$

$$y = y_{off} + m \times d + \frac{d}{2} \tag{5.4}$$

$$n = \frac{(x - x_{off} - \frac{d}{2})}{d} \tag{5.5}$$

$$m = \frac{(y - y_{off} - \frac{d}{2})}{d} \tag{5.6}$$

À medida que as coordenadas métricas dos centróides são convertidas em posições do tabuleiro é avaliada se a posição corresponde a uma posição válida de jogo. Inicialmente, verifica-se em que linha é que se encontra a peça, seguindo-se a análise da sua posição na coluna. Para manter uma organização, criou-se uma numeração do tabuleiro, tendo a sua origem no canto superior esquerdo com a numeração 0,0 e a sua finalização no canto inferior direito 7,7. Desta forma, e como as regras do jogo o definem, o jogo só se procede nos quadrados pretos. Por exemplo, na primeira linha as colunas válidas são as pares (0, 2, 4, 6) enquanto que na segunda linha só são validadas as colunas ímpares (1, 3, 5, 7). A distribuição instantânea das peças do tabuleiro é guardada numa matriz, tomando diferentes valores e disposições consoante as disposições reais do jogo. As figuras 5.20a e 5.20b representam o resultado da construção do tabuleiro virtual das situações das figuras 5.15a e 5.15b, respetivamente. Nas figuras verifica-se que a numeração corresponde à posição das peças no tabuleiro real, em ambas as situações.



(a) Disposição inicial do jogo.

(b) Disposição do jogo corrente.

Figura 5.20: Tabuleiro virtual com a representação de duas situações de jogo.

A mensagem enviada para o `algoritmo de jogo` funciona sobre o modelo cliente servidor. Este nodo ROS calcula a jogada do robô, com base na disposição do tabuleiro que recebe.

Capítulo 6

Aplicação ilustrativa

Após se tratar a informação recebida do sensor Kinect e se ter criado o tabuleiro virtual com a disposição das peças do jogo, o computador prepara-se para calcular as trajetórias do robô e enviar a informação para o servidor que está localizado no controlador do robô. O servidor analisa a informação recebida e executa-a no robô (figura 6.1). Este capítulo foca-se nos procedimentos adotados para concretizar a aplicação ilustrativa, isto é, estudar as jogadas mediante a disposição instantânea do tabuleiro, calcular as trajetórias e executá-las no robô real.

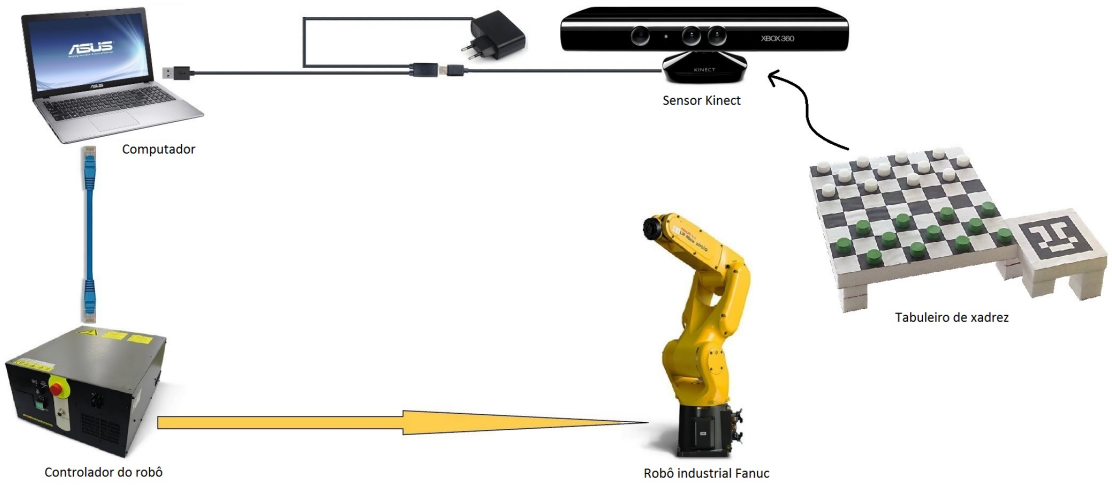


Figura 6.1: Esquema de utilização do hardware.

A sequência de explicação deste capítulo inicia com o algoritmo de jogo, que depende da informação do capítulo anterior. Depois dá-se continuidade à explicação da função tratamento que culmina no envio das coordenadas de posição para o nodo posição robô.

Por fim, apresenta-se o jogo de damas no qual se demonstra a sequência de movimentos para cumprir uma jogada do robô.

6.1 Algoritmo de jogo

O `algoritmo de jogo` funciona como servidor num modelo cliente/servidor. A estrutura base do algoritmo estava disponível em [40]. A estrutura de programação estava disponível em *open source*, sendo que o autor não concluiu a sua implementação, deixando sobretudo a inteligência artificial por completar. Avaliou-se e adaptou-se a estrutura de programação do autor para que se pudesse não só implementar as regras do jogo para executar a aplicação demonstrativa, mas também o seu nível de robustez para que não ocorressem falhas durante o cálculo da jogada.

A estrutura adaptada do algoritmo está presenta na figura 6.2. O nodo inicia com a análise da mensagem proveniente do cliente. A mensagem recebida corresponde à matriz do tabuleiro virtual, enquanto que a resposta enviada pelo servidor é composta por três termos, a origem, o destino e o remover.

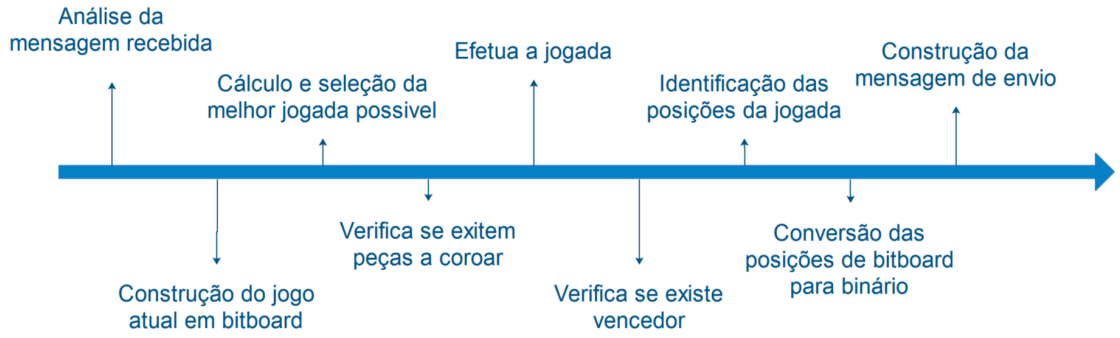


Figura 6.2: Esquema temporal do funcionamento do nodo de algoritmo de jogo.

Com base na análise da mensagem recebida, construiu-se um novo tabuleiro virtual para que fosse analisada a jogada do robô. Uma vez que o algoritmo estava desenvolvido em formato *bitboard*, converteu-se a matriz recebida de binário para formato *bitboard*. *Bitboard* é uma representação do tabuleiro em hexadecimal. Na construção do tabuleiro original, diferenciaram-se as peças peão das peças rainhas.

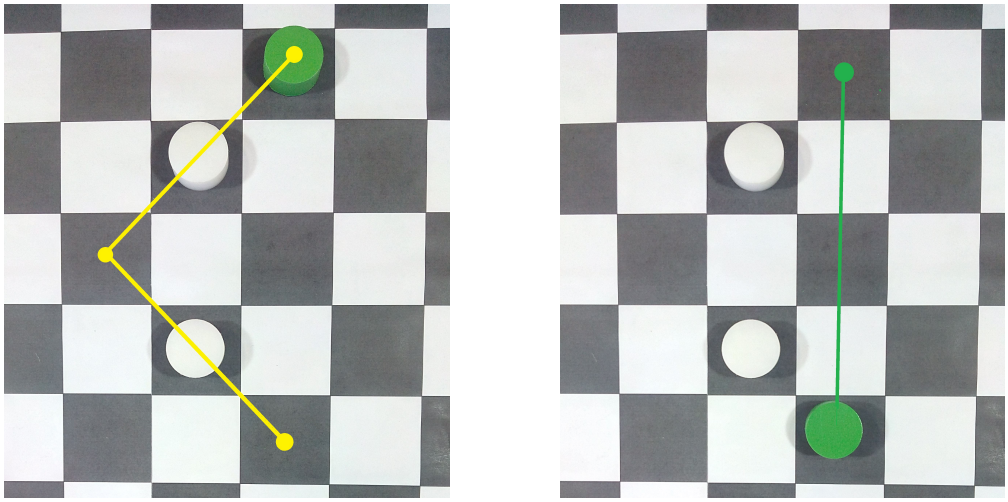
De seguida, analisou-se a existência de uma jogada válida. Para que fosse possível esta análise, criou-se um segundo tabuleiro onde foram simuladas todas as jogadas. A simulação de todas as jogadas possíveis da atual disposição das peças foi feita com recurso à função `bestMJalt`. Esta função aceita como argumento o tabuleiro real, o segundo tabuleiro e o nível de dificuldade, devolvendo o melhor movimento válido. O primeiro nível de dificuldade garante que todas as regras são cumpridas e que não ocorrem erros durante o jogo. Por sua vez, o segundo nível de dificuldade acrescenta o estudo de jogadas futuras. Dado que o código original continha gralhas, promoveu-se o seu aperfeiçoamento de forma a ser utilizado no primeiro nível de dificuldade. No entanto, o código ficou apto para, quando fosse implementado o segundo nível de dificuldade, este fosse facilmente integrado.

De seguida, executa-se o movimento da peça no tabuleiro original. Antes de se proceder ao movimento verifica-se se o mesmo inclui ou não um salto por cima de uma peça. No caso de um movimento simples ou de um movimento com salto são utilizadas duas funções distintas, `makeMove` e `makeJump` respetivamente. Ambas as funções recebem como argumentos de entrada o tabuleiro e o índice do melhor movimento válido.

No interior das funções `makeMove` e `makeJump` são adquiridas as três posições para uma jogada. Por exemplo, o cálculo da posição de origem é feita com a análise dos dois tabuleiros, o original e o tabuleiro com a jogada já realizada. Utilizando a operação `XOR` entre os dois tabuleiros, facilmente se encontra a posição onde os valores são diferentes entre si. A mesma metodologia foi adaptada para localizar as posições de destino e remover. Posteriormente, converteu-se todas as posições de *bitboard* para binário.

Uma vez que a posição remover nem sempre é uma posição a utilizar numa jogada, esta variável está por defeito com o valor (-1). Toma o valor da posição quando é necessário remover-se uma peça do tabuleiro.

Como as posições da jogada a realizar pelo robô já se encontram definidas, a última tarefa deste nodo ROS é o envio das três posições para o cliente ROS. No entanto, as três posições, correspondentes às posições de *pick*, *place* e possibilidade de remover uma peça do adversário, não são suficientes para situações de jogo particulares. Duas das situações a destacar é em casos de movimentos compostos e de coroação pertencente ao robô. A figura 6.3a contempla a situação inicial de um movimento composto enquanto que a figura 6.3b representa o fim do movimento da peça. Na situação de coroar uma peça pertencente ao robô, a substituição da peça peão para a peça dama fica a cargo do utilizador humano.



(a) Movimento composto teórico. (b) Movimento composto realizado.

Figura 6.3: Disposições inicial e final de um movimento composto.

6.2 Continuação da `função tratamento`

A receção da mensagem enviada pelo `nodo algoritmo de jogo` promove a continuidade da execução da `função tratamento`, a qual apresenta os seus objetivos na figura 6.4 .

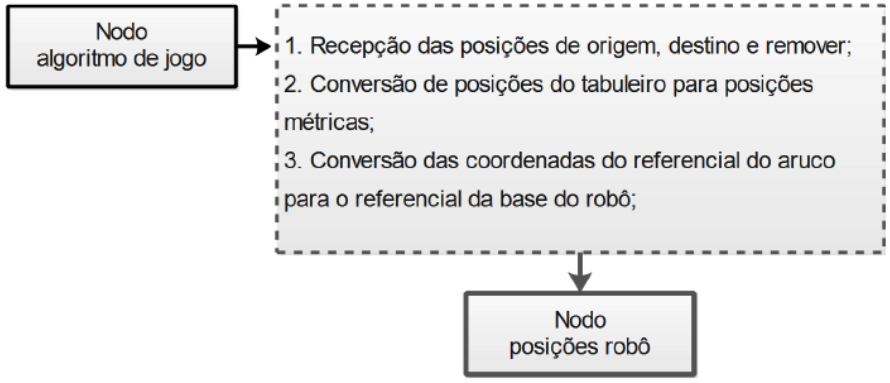


Figura 6.4: Detalhe da programação da função tratamento depois do algoritmo de jogo.

A mensagem recebida é composta pelos três termos, o termo origem, o termo destino e o termo remover. Cada um destes termos é composto por um número de dois dígitos, sendo que o primeiro dígito corresponde à coluna e o segundo dígito corresponde à linha do tabuleiro onde está a peça a movimentar. O único termo em que o primeiro dígito pode não corresponder a uma posição do tabuleiro é o termo remover.

Dado que se pretende posicionar o robô nas posições estipuladas pelo algoritmo, calculou-se a transformação geométrica entre os sistemas de coordenadas da base do robô e da coordenada da peça. Isto é, a transformação geométrica pretendida é ${}^{R}T_{P}$ da equação 4.1. Como o termo ${}^{R}T_{K}$ já foi determinado, falta calcular o termo ${}^{K}T_{P}$, realizado com base na equação 4.3.

Para converter as posições do tabuleiro em coordenadas métricas, usam-se as equações 5.3 e 5.4. O resultado das equações anteriores corresponde às posições da peça em relação ao sistema de coordenadas do *aruco marker* (${}^{A}T_{P}$). Com esta transformação geométrica e com a transformação geométrica da posição do sensor em relação ao *aruco marker* (${}^{K}T_{A}$), calcula-se o termo desconhecido (${}^{K}T_{P}$).

Em suma, a equação 4.1 é representada pela equação 6.1

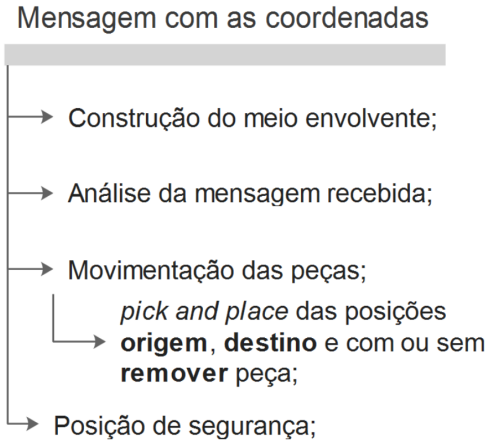
$$ {}^{R}T_{P} = {}^{R}T_{K} \times ({}^{K}T_{A} \times {}^{A}T_{P}) \tag{6.1} $$

${}^{R}T_{P}$ determina a transformação geométrica a aplicar nas coordenadas, por exemplo, as de uma peça de forma a ser aplicada no sistema de coordenadas do robô. No entanto, as coordenadas até ao momento correspondiam à posição no plano. Para determinar a altura z de modo a alcançar a superfície da peça, considerou-se uma altura constante, 0.132 mm. Esta consideração é válida, pois o tabuleiro é sempre paralelo ao plano da mesa.

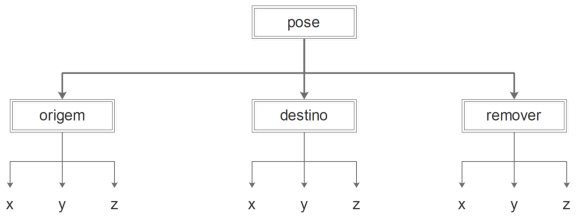
A mensagem trocada com o nodo posição robô está presente na secção 6.3.

6.3 Nodo posições robô

Este nodo ROS calcula os movimentos que o manipulador terá de percorrer, entre a posição atual e uma posição objetivo. Contém a estrutura da figura 6.5a onde, após a receção de uma mensagem nova, constrói, virtualmente, os elementos do meio envolvente, analisa a mensagem recebida, movimenta as peças e retorna à posição de segurança. A mensagem recebida tem o formato da figura 6.5b. A mensagem trocada foi desenvolvida de forma a conter a informação para todas as posições. Foi criada uma mensagem estruturada onde, numa só mensagem, são enviadas as 9 coordenadas de posição.



(a) Esquema temporal.



(b) Estrutura da mensagem recebida.

Figura 6.5: Esquema temporal dos acontecimentos que ocorrem no presente nodo e a estrutura da mensagem recebida.

Os modelos do meio envolvente do robô são incluídos no simulador e são: a bancada de trabalho, o sensor e a nuvem de pontos captada pelo sensor (figura 6.6). As dimensões e a posição dos modelos foram feitas com base na dimensão e na posição real.

Seguidamente, é executado um conjunto de funções para definir as posições. A figura 6.7 contempla a sequência das funções. Para cada posição do tabuleiro, são definidos três movimentos: um de aproximação, um que corresponde à posição da peça de jogo e um segundo de aproximação.

Cada uma das funções inicia com a especificação dos parâmetros da classe group [29]. As funções especificadas no trabalho foram as seguintes:

- definir o elo do *end-effector* (setEndEffectorLink);
- definir o tempo de planeamento (setPlanningTime);
- definir o número de vezes que é calculado o trajeto até ser devolvida uma solução (setNumPlanningAttempts);

Após se definirem os parâmetros, ajusta-se a orientação do *end-effector*. A orientação do *end-effector* é o resultado das transformações que ocorrem entre este o o sistema de coordenadas da base. Como a inclinação do tabuleiro e das peças é sempre perpendicular ao plano da mesa, realizou-se uma rotação de π radianos em torno do eixo das abcissas

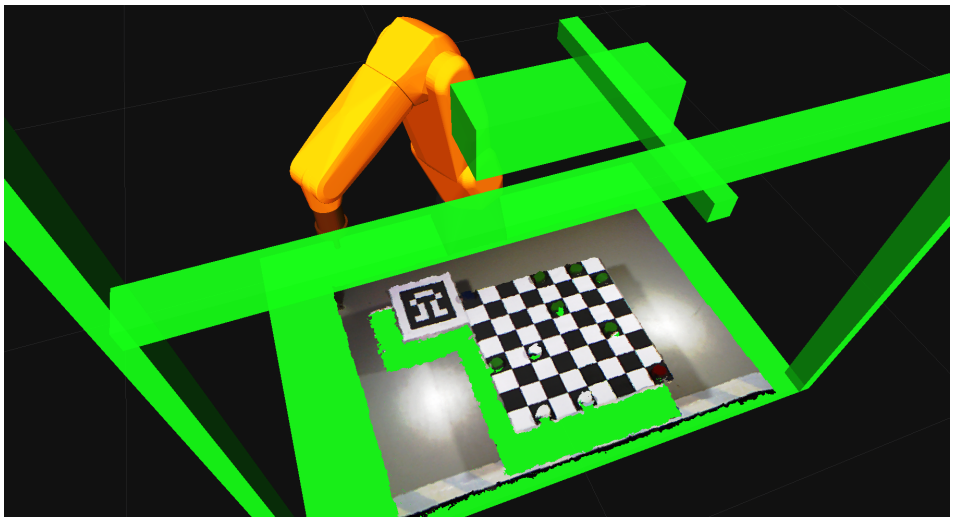


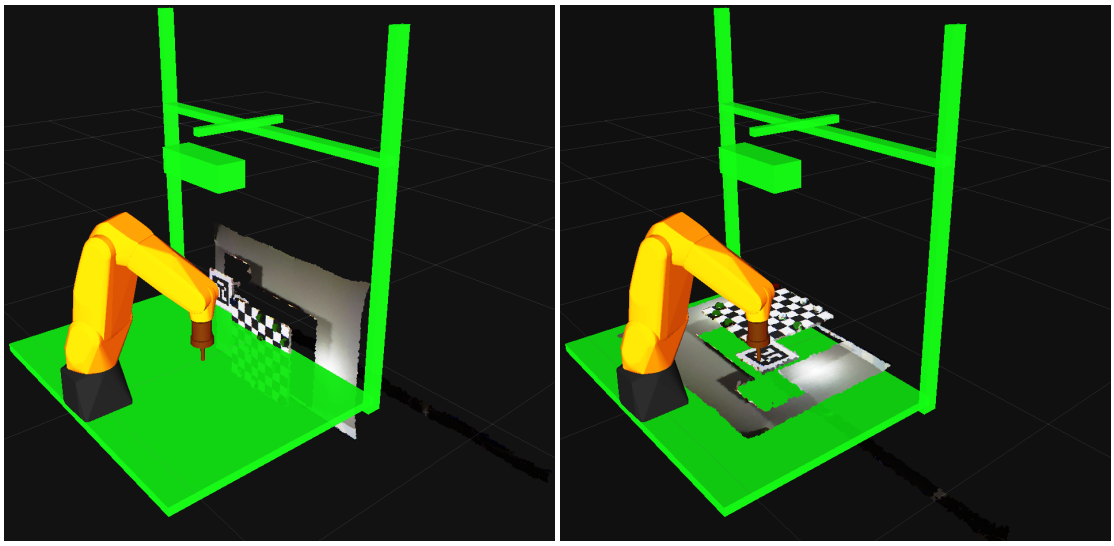
Figura 6.6: Disposição dos modelos no simulador RViz.

Figura 6.7: Esquema do grupo de funções que definem as posições.

da base, de forma a que o *end-effector* ficasse perpendicular à mesa de trabalho. O procedimento adotado foi o utilizado na secção 4.3.4.

Seguidamente, realizou-se um planeamento da trajetória entre a posição atual e a posição objetivo, com a orientação sempre perpendicular à mesa de trabalho. As várias posições que o *end-effector* do robô ocupa dependem das coordenadas recebidas. Por último, com recurso à função `group_move`, o trajeto visualizado RViz é enviado para o servidor que está em execução no controlador do manipulador real.

Para que a representação no RViz fosse o mais realista possível, aplicou-se uma transformação geométrica aos pontos recolhidos pelo sensor. Por defeito, a posição e a orientação deste sistema de coordenadas é coincidente com o sistema de coordenadas da base do sensor. A transformação aplicada ao sistema de coordenadas do sensor é a matriz de transformação ${}^{R}T_{K}$, semelhante às matrizes da figura 4.32. A figura 6.8a e a 6.8b representam a transformação do sistema de coordenadas. De salientar que esta transformação é meramente representativa.



(a) Antes da transformação. (b) Depois da transformação.

Figura 6.8: Representação do ambiente simulado RViz com a representação antes e depois da transformação do sistema de coordenadas do sensor.

6.4 Jogo de damas em ambiente ROS-Industrial

Esta secção apresenta o resultado das metodologias criadas neste trabalho que culminam num jogo de damas, validando assim a API. Baseada na metodologia por imagem 2D (figura 5.12), criou-se um método cíclico (figura 6.9) onde é executado todo o procedimento entre a receção da imagem até ao envio das coordenadas para o nodo posições robô (retângulo amarelo).

A figura 6.9 contém três regiões: a região delimitada pelo retângulo amarelo, a região de decisão e a região de execução. A primeira representa a metodologia por imagem 2D (explicada anteriormente), onde é executado todo o procedimento entre a receção da imagem até ao envio das coordenadas para o nodo posições robô.

No momento do envio da mensagem para o nodo posições robô surge a região de decisão. Esta define se a mensagem é enviada para o nodo posições robô, de forma a executar a jogada, ou se o processo de análise é reiniciado. Ao reiniciar o processo de análise implica que se execute novamente o procedimento desde a receção da imagem até ao envio das coordenadas. A região de decisão é controlada pelo nodo auxiliar teclado que aguarda que a tecla *enter* seja pressionada. A tecla ao ser pressionada, corresponde à vez do robô, o que implica que o nodo posições robô seja executado.

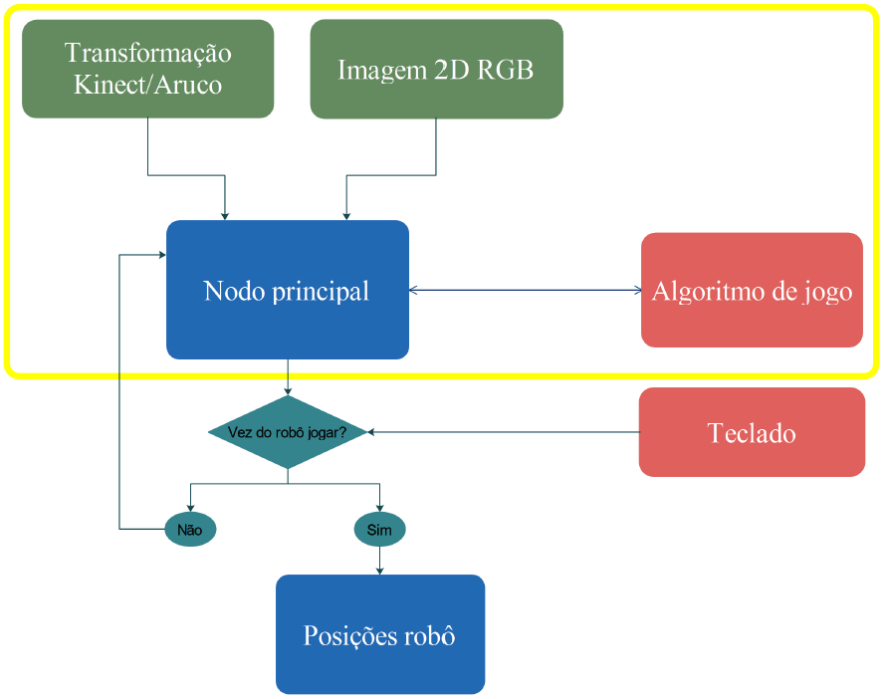
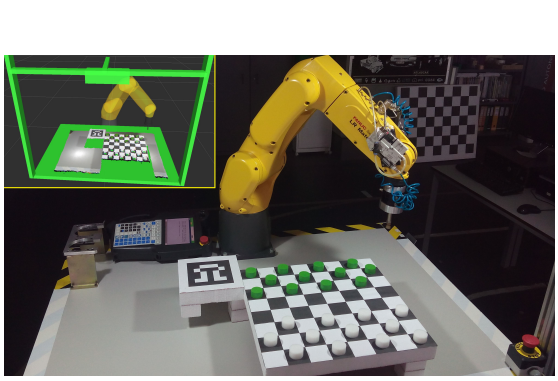


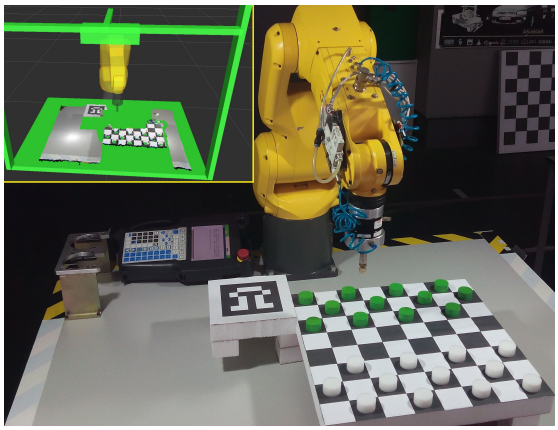
Figura 6.9: Diagrama de funcionamento da jogada do robô. Contempla os tópicos (a verde), os nodos principais (a azul) e os nodos auxiliares (a vermelho).

De seguida, são apresentados dois grupos de imagens correspondentes a duas situações de jogo distintas. O primeiro grupo de imagens está presente nesta secção, enquanto que o segundo se encontra no apêndice C. Esta última secção representa uma situação de jogo em que a sequência de movimentos contém a remoção de uma peça do adversário.

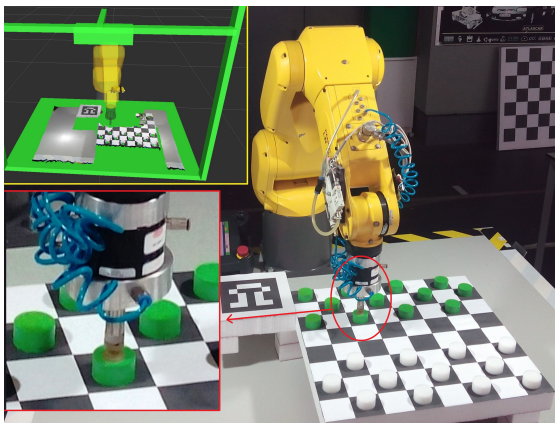
No início de um jogo de damas, estão dispostas as 24 peças no topo do tabuleiro e a posição do robô na posição inicial (figura 6.10a). Após se recolher a disposição das peças, calcula-se a jogada que o robô terá de efetuar e converte-se para o sistema de coordenadas do robô. O manipulador dá inicio à execução do grupo origem da figura 6.7. A primeira posição a ocupar é a função `approach_pick` que corresponde à posição de segurança de *pick* (figura 6.10b). Nesta o *end-effector* do robô encontra-se no plano cartesiano do centro físico da peça a mover, mas numa coordenada z acima da superfície real da peça. A posição seguinte (figura 6.10c) corresponde à função `pick` e é onde o robô se posiciona na superfície da peça, o ar comprimido é acionado e a peça é anexada ao robô. Por fim, retorna-se à posição de segurança de *pick* (figura 6.10d) e aguarda-se pela ordem de movimento para recolocar a peça no tabuleiro.



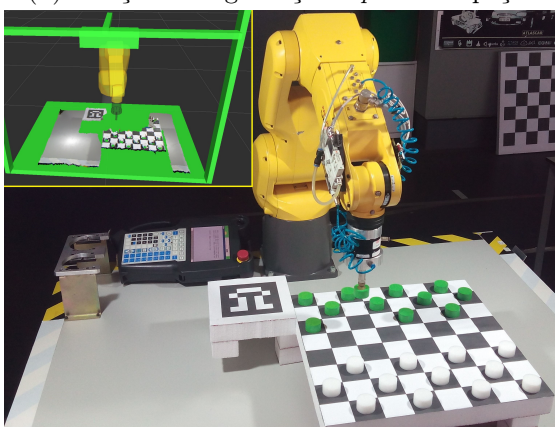
(a) Posição inicial.



(b) Posição de segurança de *pick* sem peça.



(c) *Pick* da peça do robô.

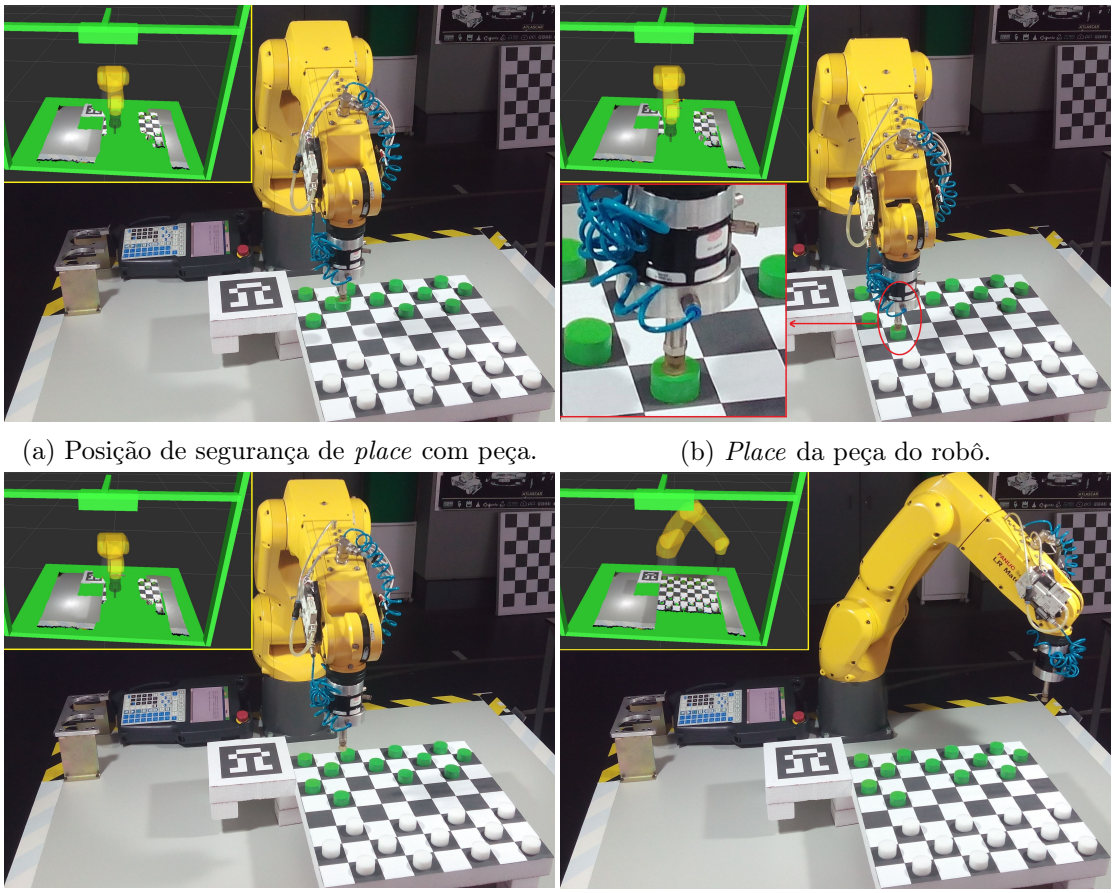


(d) Posição de segurança de *pick* com peça.

Figura 6.10: Etapas do movimento que compõem o *pick* da peça selecionada. Os retângulos amarelo e vermelho representam a visualização RViz e o detalhe, respetivamente.

Como o robô se encontra na posição de segurança de *pick*, recorre-se à função `approach_place` que corresponde à posição de segurança de *place* (figura 6.11a). Nesta

posição o *end-effector* do robô encontra-se no plano cartesiano do centro físico da peça a mover, mas numa coordenada z acima da superfície real da peça. A posição que se segue corresponde à função `place` (figura 6.11b), a qual posiciona a peça na superfície do tabuleiro. Nesta posição o ar comprimido é desligado e a peça é libertada. Por último, o robô retorna às posições de segurança de *place* (figura 6.11c) e inicial (figura 6.11d). Esta sequência de movimentos corresponde ao grupo destino da figura 6.7.



(a) Posição de segurança de *place* com peça.

(b) *Place* da peça do robô.

(c) Posição de segurança de *place* sem peça.

(d) Posição inicial.

Figura 6.11: Etapas do movimento que compõem o *place* da peça selecionada. Os retângulos amarelo e vermelho representam a visualização RViz e o detalhe, respetivamente.

Capítulo 7

Conclusões e trabalho futuro

No contexto deste trabalho, adaptou-se e implementou-se o servidor ROS-Industrial para o controlador R-30iB Mate do robô industrial Fanuc LR Mate 200iD e a API ROS-Industrial. A informação que o servidor troca com o cliente e todos os movimentos realizados no robô são realizados no espaço de junta, a uma velocidade constante. Tal como o controlo da velocidade, o controlo da força também não é suportado pelo servidor. O ROS-Industrial e em particular o ROS Fanuc demonstrou-se uma alternativa moderna que permite desenvolver aplicações complexas com um elevado nível de abstração dos aspetos físicos e de programação. O ROS-Industrial combinado com as vantagens fornecidas pelo ROS e o MoveIt! permite, entre diversas possibilidades, a interação com o robô de forma a:

- planear movimentos;
- criar os modelos virtuais do robô;
- incluir objetos presentes no meio envolvente do robô no cálculo de trajetórias;
- obter dados, tais como cinemáticas e jacobiano;
- transferir programas TP desenvolvidos no computador para o controlador;
- prevenir colisões entre os elos do próprio robô e de objetos presentes na sua área de trabalho;

Determinou-se, a cada jogada do robô e de forma automática, a transformação geométrica da base do robô para cada uma das peças que se pretendia movimentar. Esta metodologia desenvolvida engloba os sistemas de coordenadas da aplicação demonstrativa e intitula-se calibração. Uma vez que se recorreu ao sensor Kinect para identificar as posições das peças, as transformações geométricas foram subdivididas em dois grupos. O primeiro permite determinar a transformação geométrica entre a base do robô e a posição do sensor. O segundo determina a posição de todas as peças em relação ao sistema de coordenadas fixo ao tabuleiro, *aruco marker*. Combinando estes dois grupos, calculou-se a principal transformação geométrica, da base do robô para a peça a mover.

Na identificação das peças na superfície do tabuleiro foram testadas duas soluções baseadas na informação do sensor Kinect, a nuvem de pontos e a imagem 2D. A primeira solução revelou-se imprecisa e ineficaz, uma vez que o cálculo do centróide das peças não

era obtido com rigor e, por diversas vezes, havia um défice de informação correspondente à disposição real das superfícies. Desta forma, recorreu-se à informação da imagem. Esta segunda solução foi implementada com sucesso fazendo com que a informação recolhida se revelasse fiável.

Desenvolveu-se uma aplicação demonstrativa de forma a demonstrar a robustez e a versatilidade da API. O jogo de damas, aplicação demonstrativa que combinou toda a implementação do ROS-Industrial com a metodologia de calibração, revelou-se um exemplo importante de cooperação e o resultado demonstra que foram atingidos todos os objetivos propostos neste trabalho. O jogo realizou-se de forma adaptativa onde o robô, a cada jogada, se adaptava à posição e orientação do tabuleiro. Na fase de recolha e movimento das peças, o robô executa sempre movimentos de aproximação às posições das peças, possibilitando uma recolha segura e eficiente. Esta aplicação demonstrou que esta metodologia pode ser adaptada a outras operações de *pick and place*.

No entanto, verificou-se que o MoveIt! não possibilita um controlo total das redundâncias do manipulador. Quando o MoveIt! calcula o trajeto entre duas posições não possibilita a escolha das redundâncias durante um trajeto. Outra situação a salientar prende-se com o tempo de cálculo de uma jogada do robô. O tempo total de processamento de uma jogada sem remover uma peça é de 24.98 segundos. O tempo total de uma jogada com remoção de uma peça é de 30.08 segundos. Focando a análise no tempo total sem remover a peça, 24.86 segundos corresponde ao nodo posições robô, que determina as posições que o robô terá de ocupar até atingir a posição final. O restante tempo é distribuído pela captação e pelo tratamento da imagem da imagem, pelo nodo principal e pelo algoritmo de jogo. A diferença de tempo entre uma jogada com e sem remoção de peça prende-se com o cálculo adicional das posições de remoção.

7.1 Trabalho futuro

Dado que o desenvolvimento deste trabalho não teve como base outros trabalhos ou projetos realizados no laboratório, encontra-se em fase embrionária e tem potencial de evoluir. De seguida, são apresentadas as evoluções consideradas relevantes.

7.1.1 Desenvolvimento da rede de campo

A rede de campo, baseada em ModBUS/TCP, será a solução simples de desenvolver e que garante o bom funcionamento do controlo dos I/O do robô.

Por outro lado, a rede de campo permite incluir, controlar e ter acesso a outros dispositivos externos, tais como o robô Fanuc M-6iB6s presente no LAR e a sensores. A inclusão de mais três sensores Kinect, a comunicação entre eles e a adaptação da mesa de trabalho proporcionará o desenvolvimento de um trabalho que, mediante a informação das nuvens de pontos dos quatro sensores, realize a reconstrução perfeita de um elemento externo. Caso o robô esteja a efetuar um movimento e, no seu espaço de trabalho, seja detetado um movimento de um braço humano, este poderá recalcular uma nova trajetória ou simplesmente parar o movimento.

7.1.2 Plataforma móvel

A adaptação do robô industrial Fanuc LR Mate 200iD numa plataforma móvel permitirá ao robô movimentar-se para locais específicos aumentando a flexibilidade de operação. Operações de *pick and place* de objetos localizados em estantes ou locais de difícil serão resolvidos. Face ao desenvolvimento atual, necessitar-se-á da incorporação do modelo do dispositivo móvel, da alteração do tipo da junta da base do robô na ferramenta assistente de configuração e a reposição dos sensores no ambiente do robô. Adicionalmente ao sensor Kinect, que identificará os objetos a manipular, sugere-se a inclusão de dois sensores laser (SICK S300) de forma a analisar o percurso da plataforma móvel.

De forma a mover a plataforma entre duas posições, aconselha-se a utilização da *package* ROS *Navigation*. Esta biblioteca contém algoritmos para navegação autónoma em robôs móveis, como por exemplo, o SLAM e o AMCL. O SLAM cria um mapa 2D a partir dos dados dos sensores e da posição que o robô vai ocupando. O AMCL foca-se em estimar a posição da plataforma móvel.

7.1.3 Migrar este trabalho para o MATLAB

Migrar e adaptar este trabalho em ambiente MATLAB permitirá não só a utilização das funcionalidades do MATLAB, mas também tirar partido da utilização de *softwares* complementares em ambiente Windows. A MATLAB tem disponível uma *toolbox* que proporciona uma conetividade entre o MATLAB, Simulink e o ROS. A *toolbox* permite utilizar as mesmas funcionalidades disponibilizadas em ambiente Linux, tais como visualizar e analisar informação de sensores, interagir com um robô real, criar simulações e desenvolver programação para *hardware*.

Bibliografia

[1] I. F. of Robotics, "World Robotics 2015 Survey - Executive Summary", *World Robotic Report - Executive Summary*, pp. 10–21, 2015.

[2] R. Cancela, "Extensão e flexibilização da interface de controlo de um manipulador robótico FANUC", Dissertação de Mestrado, Universidade de Aveiro, 2007.

[3] T. Lozano-Perez, "Robot programming", *Proceedings of the IEEE*, vol. 71, nº 7, pp. 829–833, 1983.

[4] B. Siciliano e O. Khatib, *Handbook of Robotics*. Springer International Publishing, 2008, pp. 1006–1627.

[5] KUKA Roboter GmbH, "KUKA System Software 5.5", KUKA Roboter GmbH, Augsburg, Germany, rel. téc., 2010.

[6] Southwest research institute and tu delft robotics institute and wolf robotics, "Scan-N-Plan", ROS Industrial, Texas, EUA, rel. téc., 2015.

[7] A. Mechatronics, *Alten mechatronics unstructured box palletizing with ros-industrial*, 2013. endereço: `http://rosindustrial.org/news/2013/11/28/alten-mechatronics-unstructured-box-palletizing-with-ros-i-and-moveit` (acedido em 03/03/2016).

[8] Intermodalics, *Intermodalics uses ros-i for palletizing application*, 2014. endereço: `http://rosindustrial.org/news/2014/3/11/intermodalics-uses-ros-i-for-palletizing-application` (acedido em 02/03/2016).

[9] FANUC CORPORATION, "FEATURES Application system Load / unload from ROBODRILL Operating space", Fanuc, Yamanashi, Japan, rel. téc., 2012.

[10] ——, "R-30iB Mate CONTROLLER - Maintenance Manual", rel. téc., 2011.

[11] N. Santos, "Bin Picking de Objetos Polimórficos Convexos usando Perceção 3D", Dissertação de Mestrado, Universidade de Aveiro, 2014.

[12] Microsoft, *Kinect for windows sensor components and specifications*, 2015. endereço: `https://msdn.microsoft.com/en-us/library/jj131033.aspx` (acedido em 07/03/2016).

[13] OpenNI, *About openni*, 2015. endereço: `http://openni.ru/about/index.html` (acedido em 07/03/2016).

[14] FreeCAD, *Visã£o geral*, 2015. endereço: `http://www.freecadweb.org/wiki/index.php?title=About_FreeCAD/pt` (acedido em 14/03/2016).

[15] G. Bradski, *The opencv library*, 2000. endereço: `http://www.drdobbs.com/open-source/the-opencv-library/184404319` (acedido em 15/03/2016).

[16] Itseez, *Opencv: about*, 2016. endereço: `http://opencv.org/about.html` (acedido em 14/03/2016).

[17] L. Joseph, *Mastering ROS for Robotics Programming*, 1st Ed., P. Publishing, ed. Packt Publishing, 2015, vol. 1, p. 689.

[18] R. B. Rusu e S. Cousins, "3D is here: point cloud library", *IEEE International Conference on Robotics and Automation*, pp. 1–4, 2011.

[19] ROS, *Ros history*, 2016. endereço: `http://www.ros.org/history/` (acedido em 21/03/2016).

[20] A. Martinez, E. Fernandez, L. Sanchez Crespo e A. Mahtani, *Learning ROS for Robotics Programming*, 2st Ed., P. Publishing, ed. Packt Publishing, 2015, vol. 1, p. 458.

[21] ROS, *Industrial*, 2016. endereço: `http://wiki.ros.org/Industrial` (acedido em 21/03/2016).

[22] ——, *Industrial_ core*, 2015. endereço: `http://wiki.ros.org/industrial_core?distro=indigo` (acedido em 24/03/2016).

[23] ——, *Simple_ Message*, 2015. endereço: `http://wiki.ros.org/simple_message` (acedido em 25/03/2016).

[24] ——, *Fanuc experimental*, 2016. endereço: `https://github.com/ros-industrial/fanuc_experimental` (acedido em 25/03/2016).

[25] ——, *Fanuc tutorials*, 2014. endereço: `http://wiki.ros.org/fanuc/Tutorials/` (acedido em 03/03/2016).

[26] G. vd. Hoorn, *Fanuc_ driver*, 2015. endereço: `https://github.com/ros-industrial/fanuc/tree/indigo-devel/fanuc_driver` (acedido em 09/03/2016).

[27] V. Lamoine, *Fanuc post processor*, 2016. endereço: `https://github.com/InstitutMaupertuis/fanuc_post_processor` (acedido em 14/06/2016).

[28] Moveit!, *System architecture*, 2016. endereço: `http://moveit.ros.org/documentation/concepts/` (acedido em 12/04/2016).

[29] I. Sucan, *Moveit::planning_ interface::movegroup class reference*, 2016. endereço: `http://docs.ros.org/jade/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1MoveGroup.html` (acedido em 02/06/2016).

[30] I. A. Şucan, *Moveit::planning_ interface::planningsceneinterface class reference*, 2016. endereço: `http://docs.ros.org/jade/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1PlanningSceneInterface.html` (acedido em 22/04/2016).

[31] I. A. Şucan, M. Moll e L. E. Kavraki, *The Open Motion Planning Library*, 2012. endereço: `http://ompl.kavrakilab.org/` (acedido em 06/05/2016).

[32] ——, *How to benchmark planners*, 2012. endereço: `http://ompl.kavrakilab.org/benchmark.html` (acedido em 06/04/2016).

[33] ROS, *Kdl*, 2014. endereço: `http://wiki.ros.org/kdl` (acedido em 14/04/2016).

[34] ——, *Ikfast plugin for moveit!*, 2015. endereço: `http://wiki.ros.org/Industrial/Tutorials/Create_a_Fast_IK_Solution/moveit_plugin` (acedido em 28/04/2016).

[35] I. A. Şucan, S. Chitta e A. Pooley, *Robot_ state::robotstate class reference*, 2014. endereço: `http://docs.ros.org/jade/api/moveit_core/html/classmoveit_1_1core_1_1RobotState.html` (acedido em 27/04/2016).

[36] V. Santos, "Robótica Industrial - Apontamentos Teóricos", Departamento de Engenharia Mecânica, Universidade de Aveiro, 2003.

[37] J. Bohren, *Aruco / visp hand-eye calibration*, 2014. endereço: `https://github.com/jhu-lcsr/aruco_hand_eye` (acedido em 02/04/2016).

[38] B. Magyar, *Marker582_ 5cm.jpg*, 2014. endereço: `https://github.com/pal-robotics/aruco_ros/blob/master/aruco_ros/etc/marker582_5cm.jpg` (acedido em 02/06/2016).

[39] M. Andersen, T. Jensen, P. Lisouski, A. Mortensen, M. Hansen, T. Gregersen e P. Ahrendt, "Kinect Depth Sensor Evaluation for Computer Vision Applications", *Electrical and Computer Engineering*, p. 37, 2012.

[40] A. Allen, *Checkers engine written in c.* 2009. endereço: `https://github.com/aronallen/checkers` (acedido em 02/04/2016).

Apêndices

Apêndice A

Procedimento da ferramenta assistente de configuração

O *launch* `roslaunch moveit_setup_assistant setup_assistant.launch` disputa o início da interface da ferramenta, gerando a janela da figura A.1. Esta janela está preparada para incluir modelos do formato URDF ou COLLADA de um braço robótico. Uma vez que os modelos já estavam construídos, fez-se o *load* dos mesmos.

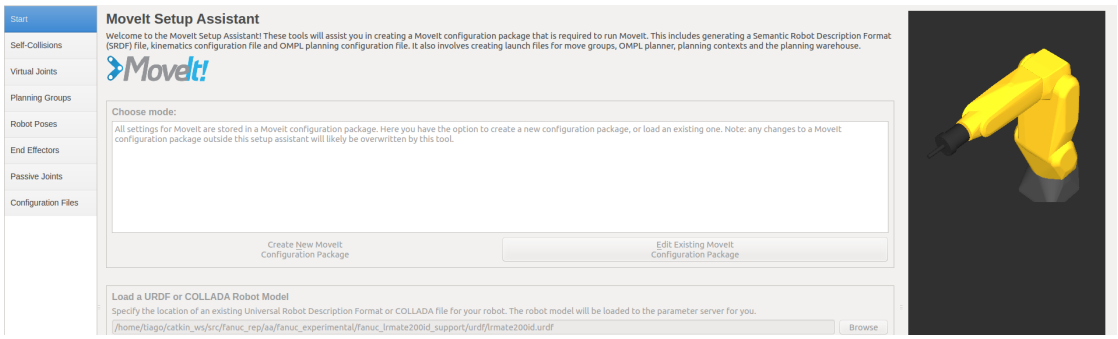


Figura A.1: Primeiro separador: *load* dos modelos do robô.

Na figura A.2, correspondente ao passo dois, é gerada a matriz ACM. O utilizador constrói a matriz de forma tão automática e a tão alto nível, que basta pressionar no botão assinalado na figura para a gerar. Este procura os pares de elos do robô que estão a salvo de colisão e desativa-os. A densidade de amostragem define a quantidade de posições aleatórias que a geometria do robô será testada para verificar se existe auto colisão. Por defeito, a densidade é 10000 verificações e garante que as principais interações são identificadas.

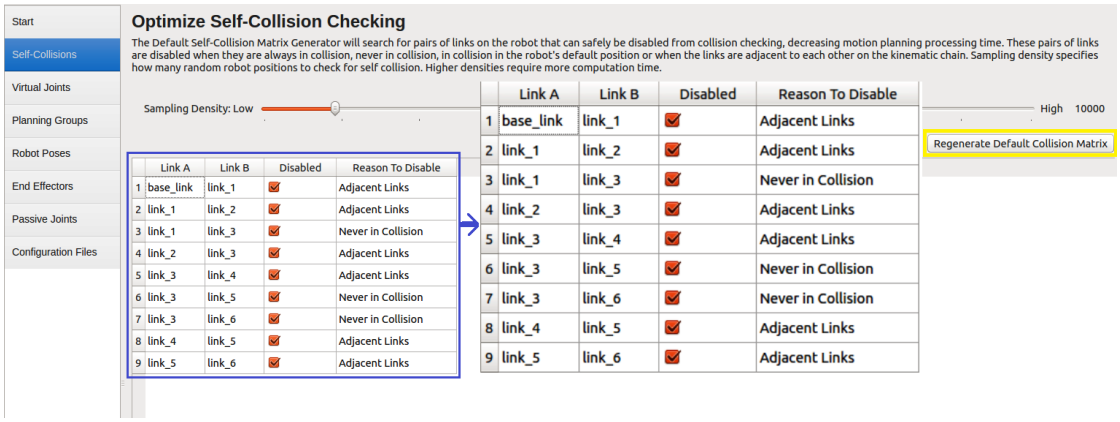


Figura A.2: Segundo separador: definição das relações entre os elos.

O próximo passo (figura A.3) tem por objetivo anexar juntas virtuais ao robô. As juntas virtuais são utilizadas nas situações em que o manipulador não é fixo. No nosso caso específico, o robô está fixo na bancada de trabalho, não tendo nenhuma junta virtual. No entanto, adicionou-se uma junta virtual para garantir que a junta fosse imóvel e de forma a ficar apto para trabalhos futuros. No caso do desenvolvimento de um trabalho com uma solução em calha amovível, bastará alterar o tipo da junta e o *parent frame*.

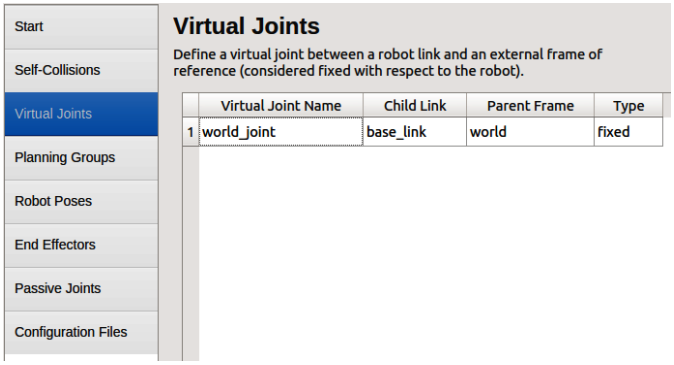


Figura A.3: Terceiro separador: Atribuição das juntas virtuais.

No quarto separador (figura A.4) define-se os grupos de planeamento. Grupos de planeamento são grupos de juntas que necessitam que o seu planeamento seja calculado em conjunto, a fim de alcançar uma determinada configuração. No caso do manipulador do trabalho, definiram-se dois grupos, o grupo para o braço e o grupo para o *end-effector*. O planeamento será então realizado em separado para definir uma posição do braço e uma posição do *gripper*.

Tal como em todas as análises em robótica, é imprescindível a especificação da posição zero ou *Home Position* (figura A.5). Definiu-se uma única posição intitulada de *home*, na qual está especificada a posição zero de fábrica do manipulador.

A quinta aba (figura A.6) é utilizada para definir a extremidade atuante, *end-effector*. No nosso caso, o *end-effector* permite sugar objetos presentes no espaço de trabalho,

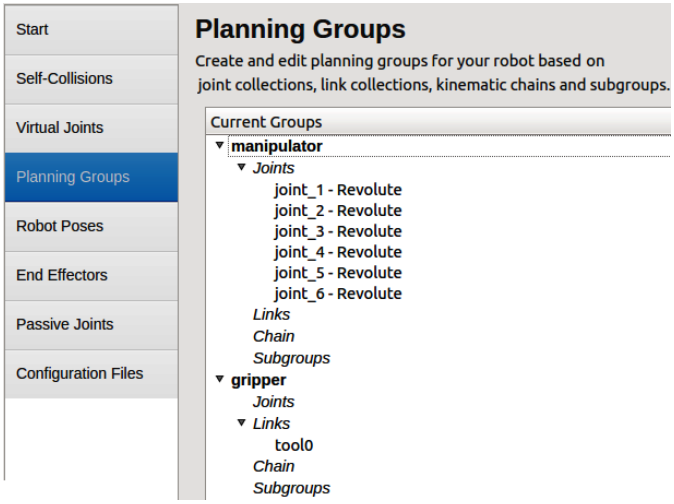


Figura A.4: Quarto separador: Grupos de planeamento.

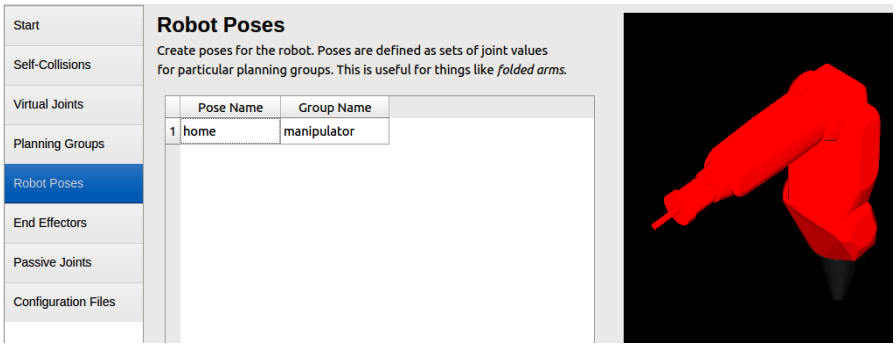


Figura A.5: Quinto separador: Definição da *Home Position*.

contendo uma ventosa.

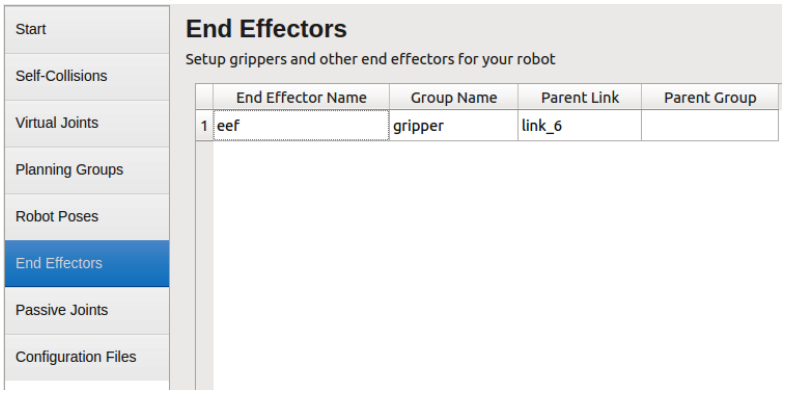


Figura A.6: Sexto separador: *End-effector*.

O sétimo separador é a etapa de configuração opcional que define as juntas que não possam ser atuadas. Um exemplo de uma junta passiva num robô poderia ser uma em que estivessem 3 juntas perfeitamente coincidentes. Desta forma, e como no trabalho não existe nenhuma junta passiva, não foram feitas alterações neste separador. Por fim (figura A.7), o último separador consiste em gerar os ficheiros padrão. Estes estão presentes na figura A.8. É necessário fornecer o caminho da *package* de configuração onde serão armazenados a maior parte dos arquivos de iniciação e configuração.

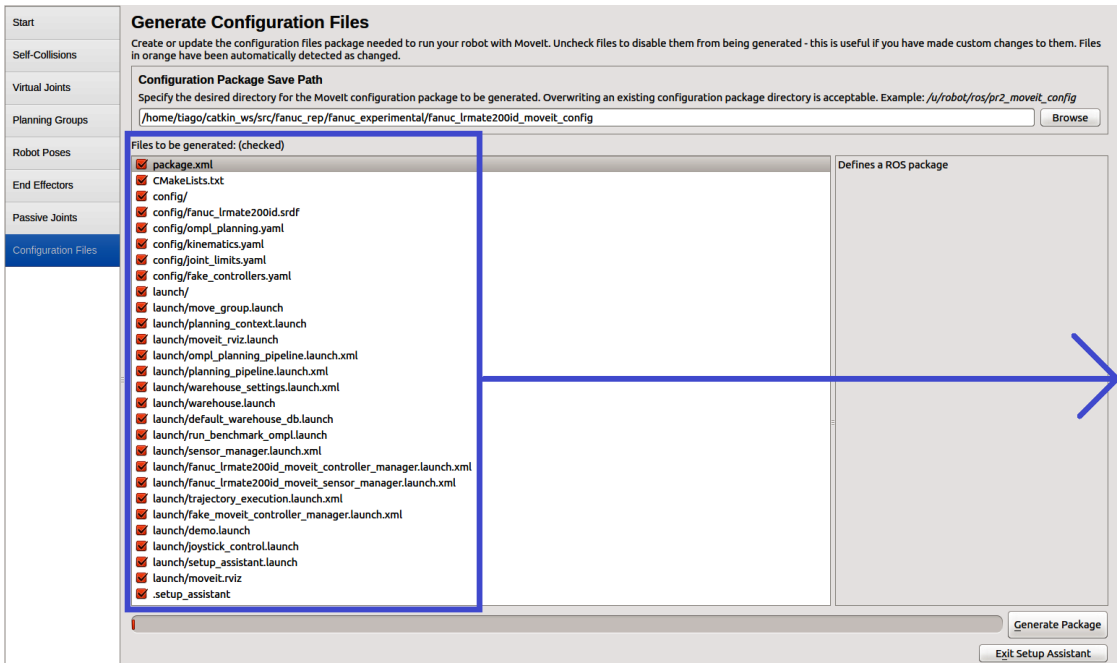


Figura A.7: Gerar e armazenar os ficheiros.

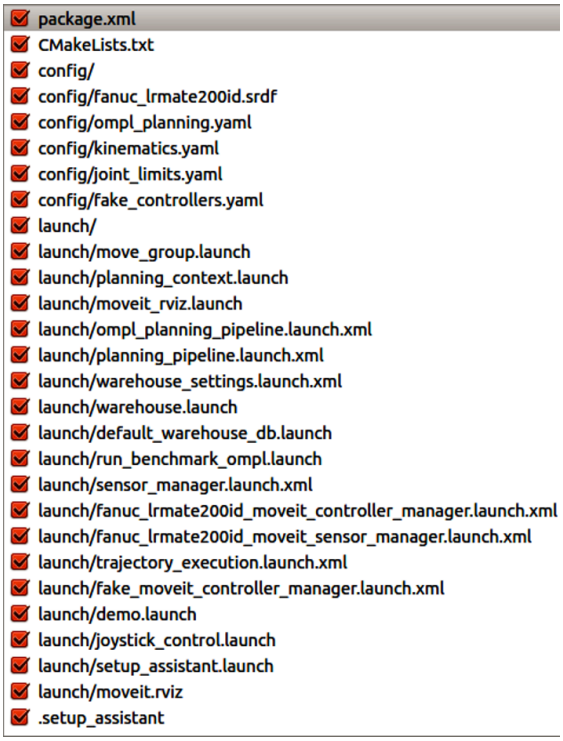
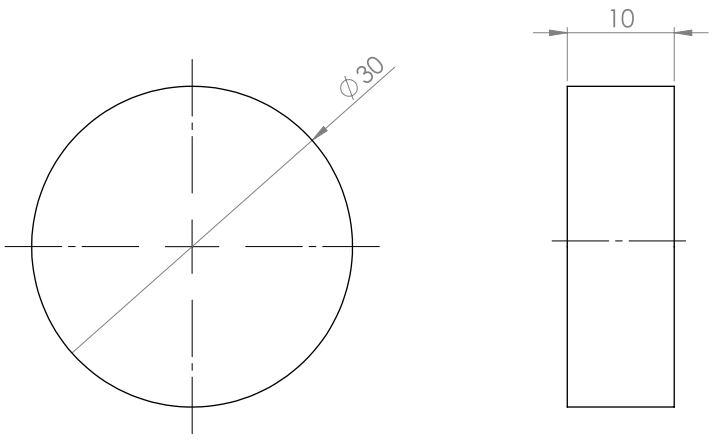


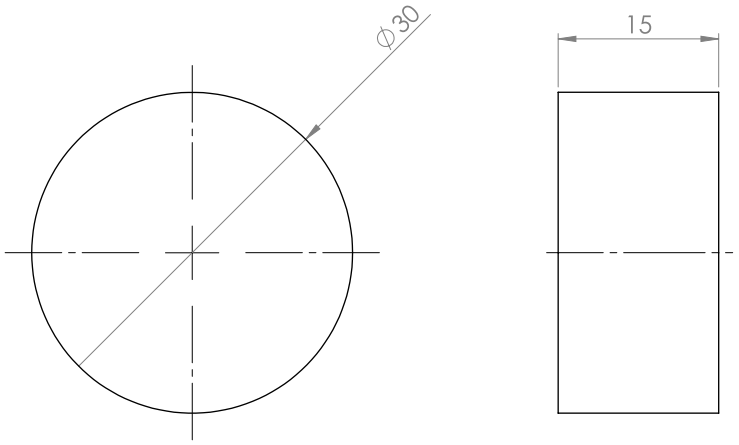
Figura A.8: Zoom da secção dos ficheiros gerados.

Apêndice B

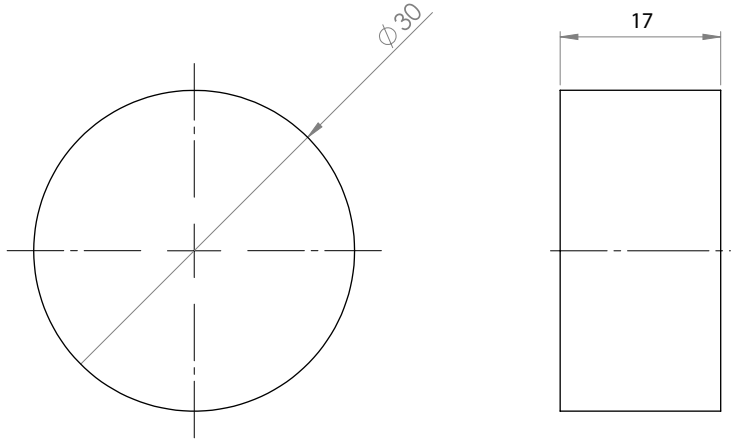
Desenhos de definição das peças de jogo e da ferramenta de calibração



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN							
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO. Peca10mm	A4
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1

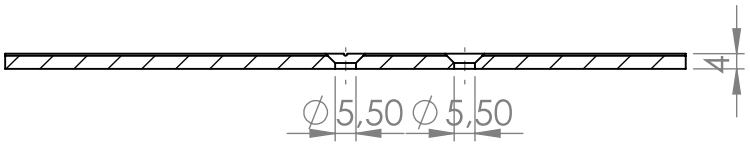


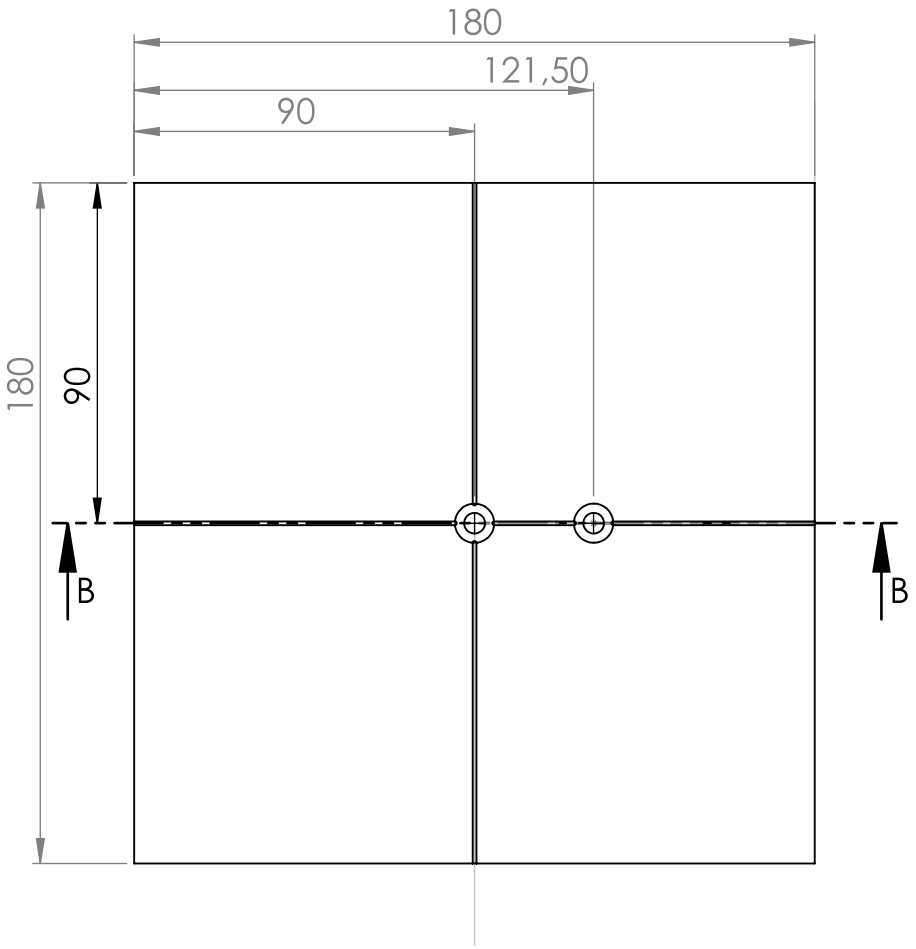
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN							
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO. Peca15mm	A4
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN							
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO. Peca17mm	A4
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1

SECTION B-B





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

MATERIAL:

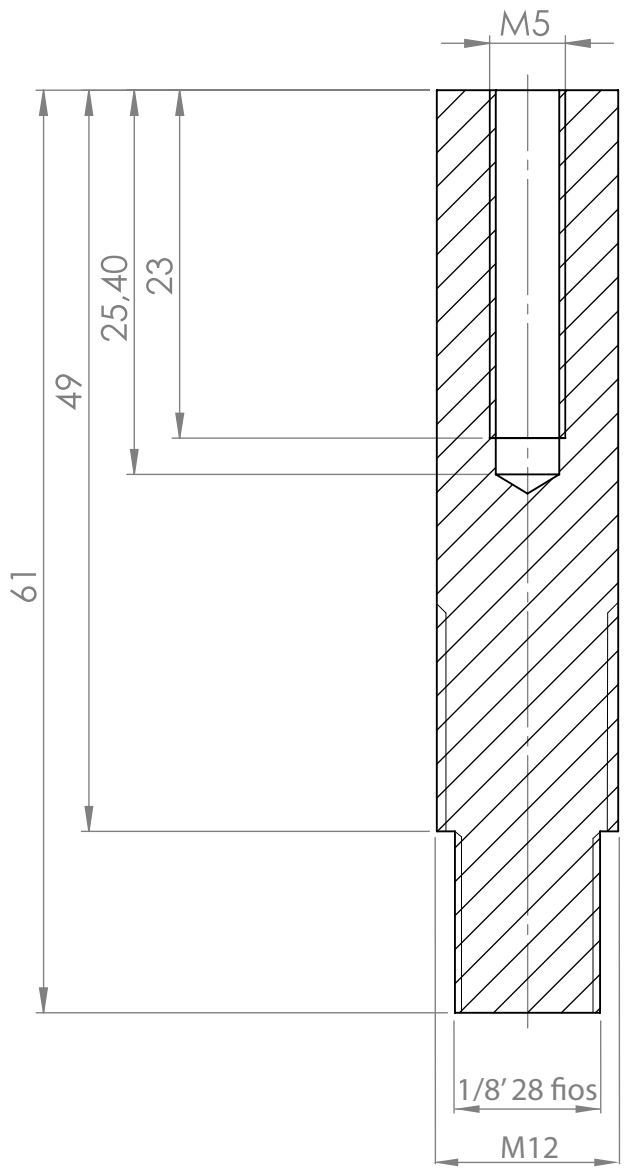
DWG NO. Chapa

A4

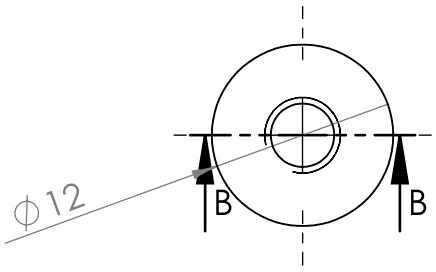
WEIGHT:

SCALE:1:2

SHEET 1 OF 1



SECTION B-B



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	REVISION

	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

MATERIAL:

DWG NO. pino

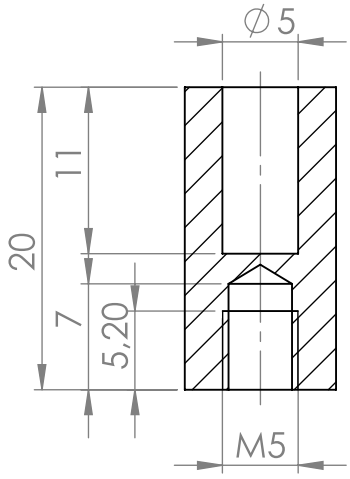
A4

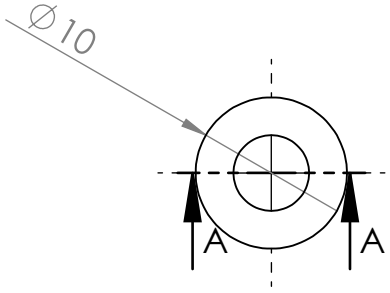
WEIGHT:

SCALE:2:1

SHEET 1 OF 1

SECTION A-A





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

MATERIAL:

DWG NO. pino_apoio

A4

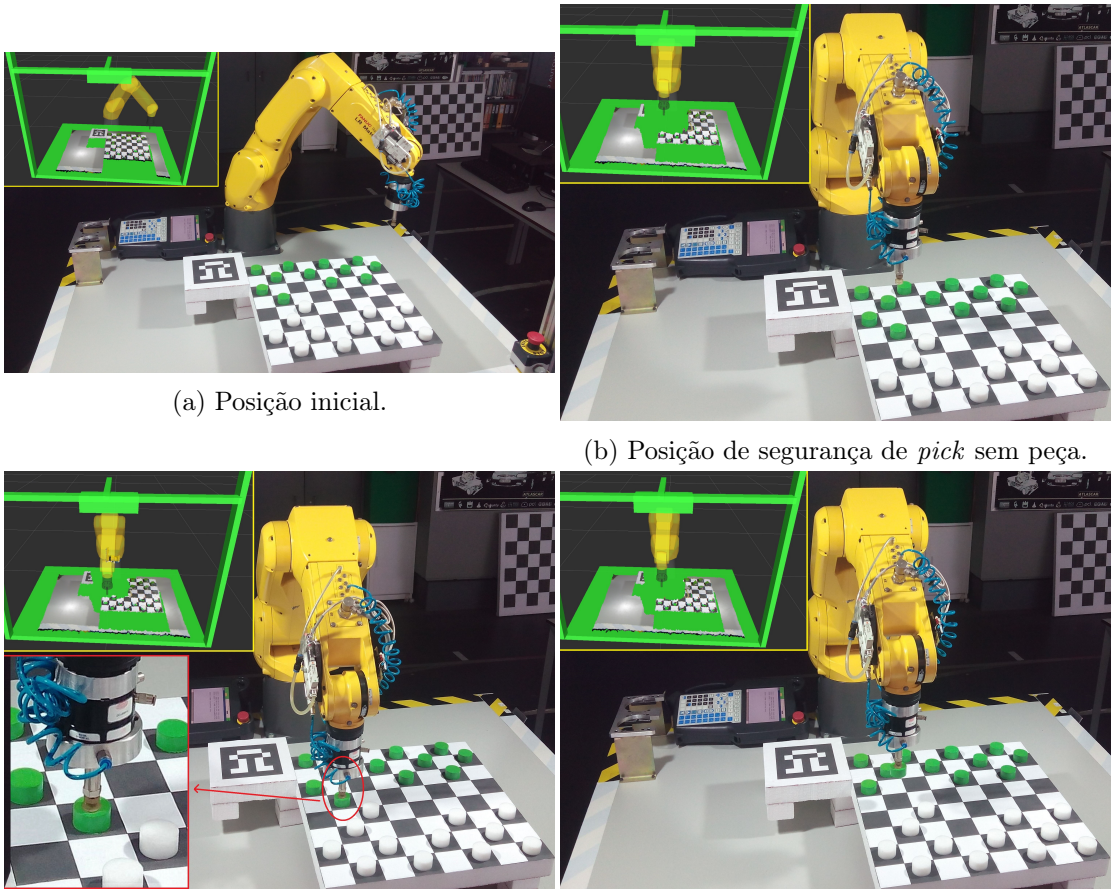
WEIGHT:

SCALE:2:1

SHEET 1 OF 1

Apêndice C

Jogada completa com remoção de peça do adversário



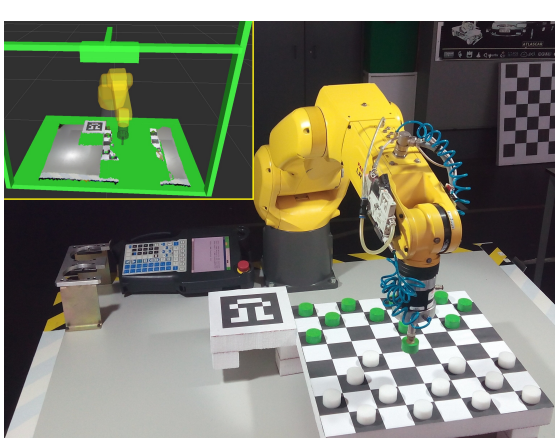
(a) Posição inicial.

(b) Posição de segurança de *pick* sem peça.

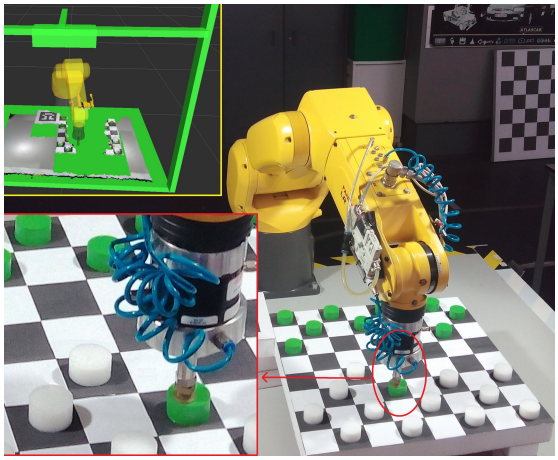
(c) *Pick* da peça do robô.

(d) Posição de segurança de *pick* com peça.

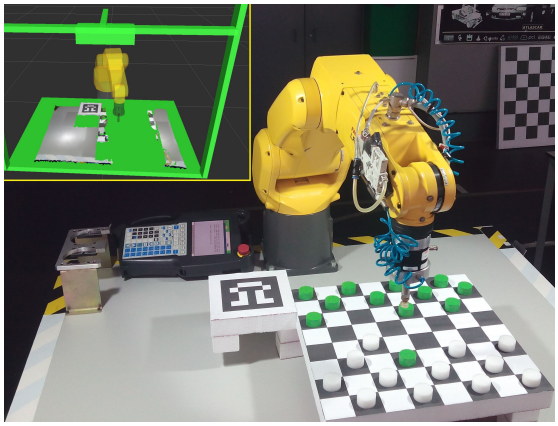
Figura C.1: Etapas do movimento que compõem o *pick* de uma peça. Os retângulos amarelo e vermelho representam a visualização RViz e o detalhe, respetivamente.



(a) Posição de segurança de *place* com peça.

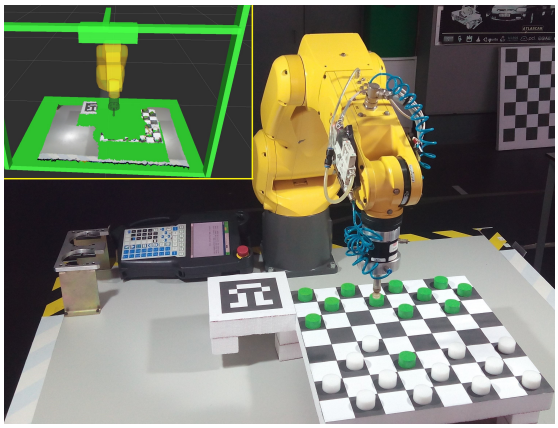


(b) *Place* da peça do robô.

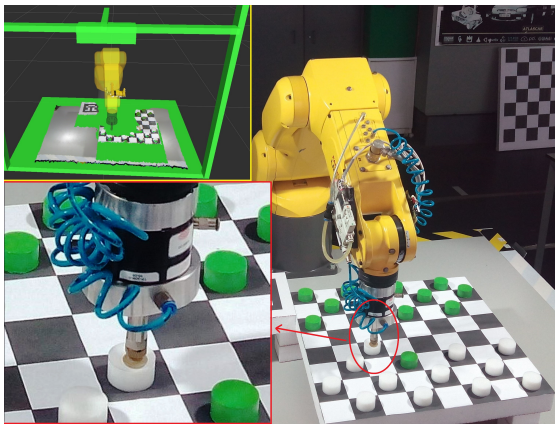


(c) Posição de segurança de *place* sem peça.

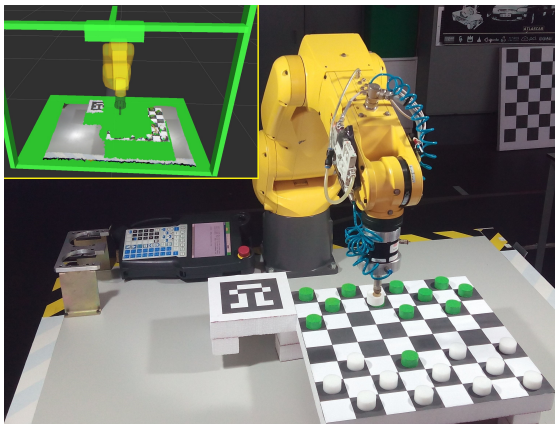
Figura C.2: Etapas do movimento que compõem o *place* de uma peça. Os retângulos amarelo e vermelho representam a visualização RViz e o detalhe, respetivamente.



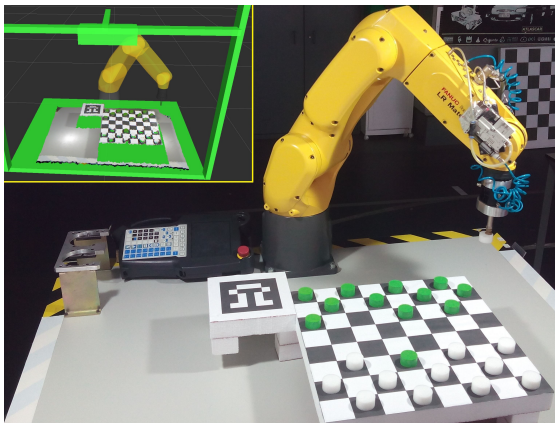
(a) Posição de segurança para remover sem peça.



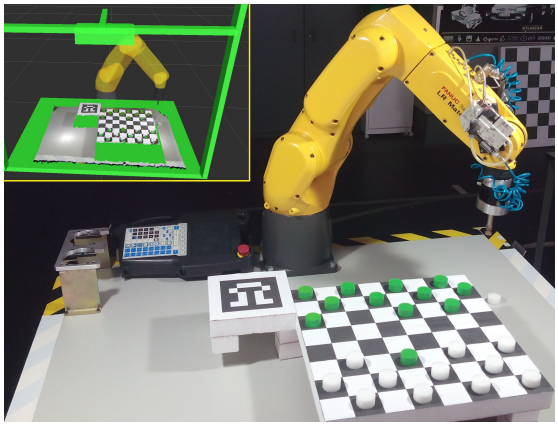
(b) *Pick* da peça para remover.



(c) Posição de segurança para remover com peça.



(d) Posição inicial com peça.



(e) Posição inicial sem peça.

Figura C.3: Etapas do movimento que compõem a remoção de uma peça do adversário. Os retângulos amarelo e vermelho representam a visualização RViz e o detalhe, respetivamente.