APLICAÇÃO DE MANIPULAÇÃO MÓVEL E ANÁLISE DE IMAGEM PARA SIMULAÇÃO DE MONTAGEM DE PARAFUSO EM SUPERFÍCIE PLANA

Larissa Cassador Casteluci

Número USP 8004645

DEPARTAMENTO DE ENGENHARIA MECÂNICA

UNIVERSIDADE DE SÃO PAULO

Junho 2018

Resumo

A utilização de Manipulação Móvel em ambientes industriais sempre apresentou um potencial grande para desenvolvimento, mas devido a diversos fatores como dependabilidade e interação robô máquina; não é concretizado. Para contornar a situação, exigem-se mais testes em simulação e em chão de fábrica. Esse projeto possui o objetivo de testar o uso em conjunto de três *softwares*, ROS, OpenCV e V-REP, para simulações nessa área.

A simulação é realizada considerando-se a montagem de parafusos em uma chapa plana por um braço mecânico apoiado em uma base móvel, com a detecção dos furos feita por meio de visão computacional. Os *softwares* OpenCV e ROS apresentaram resultados conforme o esperado, porém o V-REP apresentou problemas de robustez. Também foi analisada a precisão dos dados de velocidades e acelerações obtidos pelo programa, obtendo-se resultados acima do esperado.

Sumário

Capítulo 1

Introdução

Uma das primeiras aplicações documentadas de manipulação robótica ocorreu em 1984, quando o robô MORO, composto de um braço mecânico em uma plataforma móvel, foi configurado para andar livremente em um chão de fábrica entregando peças e ferramentas [1]. Após décadas de avanço nessa área, ela cresceu a ponto de ser necessário subdividi-la. Tradicionalmente, essa divisão ocorre em ambiente estruturado e não-estruturado, de acordo com o conhecimento prévio que o manipulador possuí sobre o ambiente.

A primeira área é a que Comitê Técnico de manipulação móvel da IEEE dá maior atenção [2]. Nela considera-se que não há ou há pouco conhecimento prévio, e portanto os focos de estudo são métodos de obtenção de informações e de como utilizá-las. Iniciativas como a DARPA *challenge* [3], e o desenvolvimento de ferramentas de SLAM (*simultaneous localization and mapping*) [4], são indicativos do avanço dessa linha de pesquisa.

Já na de área ambiente estruturado, há mais informações prévias sobre o ambiente, e portanto métodos que focam em navegação e desvio de

obstáculos se tornam menos necessários. Mas para ambientes de chão de fábrica há ainda outros critérios que sistemas robóticos devem satisfazer, mas que na manipulação móvel ainda não atingiram a maturidade necessária para seu uso eficiente. Para a utilização em grande escala dessa tecnologia na indústria, problemas de dependabilidade, normas de segurança, interação robô-humana e padronização [1], ainda precisam ser trabalhados, e portanto são necessários mais estudos e testes em simulação e no chão de fábrica.

Esse projeto busca fazer um estudo preliminar sobre uma possível implementação em ambiente industrial, simulando montagem de parafusos em uma chapa plana, com o objetivo de testar ferramentas, o seu uso em conjunto e a dependabilidade do sistema como um todo, para futuras implementações em projetos mais complexos em manipulação móvel.

Capítulo 2

Materiais

O projeto utilizou 3 softwares em conjunto. O V-REP, para a realização da simulação, incluindo os cálculos de física e de cinemática direta e inversa; o OpenCV, para o sistema de visão, e o ROS, que garantiu a comunicação entre esses dois softwares. As próximas seções irão descrever cada programa de maneira mais profunda . Um roteiro prático de como configurar e utilizar o ROS em conjunto com o V-REP está no Apêndice A.

2.1 ROS

ROS, ou *Robot Operating System*, é um conjunto de *frameworks* que garante uma grande quantidade de funcionalidades pertinentes à robótica, e em particular, à robôs móveis. A capacidade deste *software* também incluí conceitos de sistemas operacionais como abstração de *hardware*, comunicação entre diversos processos, gestão de pacotes e integração com diversas outras ferramentas, como o OpenCV. Devido à sua grande flexibilidade e parti-

cipação ativa da comunidade, seu uso tornou-se frequente para controle em sistemas robóticos.

O sistema funciona em uma arquitetura de grafos, na qual os processos, denominados de nós, comunicam-se entre si por meio de mensagens que passam por rotas previamente estipuladas, chamadas de tópicos. Para cada tópico, é estabelecido um nó editor que envia mensagens e um assinante que as recebe. O sistema é classificado como fracamente acoplado, e por causa disso e necessário o uso de um *master*. Uma das responsabilidades do *master* é garantir que os dois nós de um tópico encontrem-se para poder trocar mensagens. Os nós de um mesmo projeto devem ser organizados em pacotes que podem conter bibliotecas, arquivos de dados e de configuração independentes.

Para ilustrar melhor o funcionamento do sistema, será utilizado o turtlesim, que é um exemplo do próprio ROS. O primeiro passo é inicializar o *master*; logo, em um terminal do linux, digita-se o comando `roscore` para executá-lo. Em seguida, em um outro terminal, usa-se o comando rosrun com a seguinte sintaxe:

```
rosrun [nome_pacote] [nome nó]
```

Nesse exemplo:

```
rosrun turtlesim turtlesim_node
```

Esse comando abre a janela indicada pela Figura 2.1.

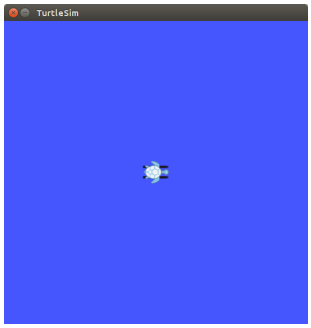

Figura 2.1: Nó turtlesim

Figura 2.2: Nó turtlesim movimentando de acordo com o teclado

Em seguida, abre-se outro nó com o nome `turtle_teleop_key`, que captura os dados das setas do teclado e os manda para o turtlesim node, movimentando a tartaruga, como mostra a Figura 2.2.

Para tornar mais fácil a visualização da comunicação do projeto, o próprio ROS dispõe de uma ferramenta que gera uma imagem demonstrando os nós ativos e os tópicos de comunicação, o pacote *ros_graph*. Para ativá-la usa-se o comando `rosrun rqt_graph rqt_graph`, que nesse exemplo gera o diagrama da Figura 2.3

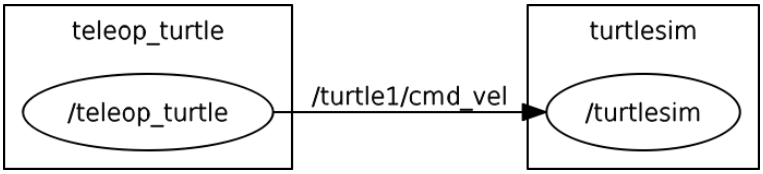

Figura 2.3: Modelo da comunicação gerado pelo ros_graph

Outra maneira possível de observar o sistema é utilizando o comando `rosnode roslist`, que lista todos os nós ativos,resultando em:

```
\rosout
\turtlesim
\teleop_turtle
```

Turtlesim e teleop_turtle são os nós citados, e rosout é o nó que desliga o *master*.

2.2 V-REP

O V-REP é um *software* voltado para a simulação de ambientes, com a possibilidade de integração de tipos de objetos, cada um controlado individualmente por meio do seu próprio *script*. Os seus principais focos são na prototipagem rápida de algoritmos, monitoramento remoto, e simulações automatizadas de fábrica.

O *software* não possui motor de fisica próprio, mas é capaz de utilizar motores de física já existentes como o Bullet, ODE e Vortex, além de ter integração com diversas ferramentas úteis para a área da robótica como a biblioteca OMPL (*Open Motion Planning Library*), que inclui diversos

algoritmos clássicos de *motion planning*, como planejadores *Rapid-exploring Random Trees* (RRT).

A pricipal característica desse programa é a sua versatilidade. Na versão atual, há 6 diferentes abordagens para a programação do código de controle dos modelos. Além disso ele incluí a possibilidade de integração com APIs remotas e de realizar operações síncronas e assíncronas.

Um ponto negativo dessa abrangência é que o *software* ainda não é muito robusto, o que pode dificultar o seu uso, e a documentação sobre as APIs é insuficiente. Esses aspectos são discutidos em mais detalhes no capítulo de resultados e discussões

Nesse trabalho, foram utilizadas a versão ROS Indigo, que é compatível com o Ubuntu 14.04. Para controle dos modelos empregou-se um sistema híbrido entre nós do ROS e *scripts* embarcados do V-REP, com as juntas do modelo operando no modo Torque, simulados com o motor de física *Bullet* e com o passo de tempo definido em 1ms. Vários desses parâmetros foram escolhidos a fim de melhorar a precisão dos resultados, e seguem a documentação. A explicação da importância e da escolha desses parâmetros estão no Apêndice A.

2.3 OpenCV

A biblioteca OpenCV é voltada para o desenvolvimento de algoritmos de visão computacional. Por ser um *software open source*, é largamente utilizado em pesquisa e na indústria.

Ela oferece algumas abstrações nas classes que definem as imagens, pro-

movendo facilidades em termos de eficiência e facilidade durante a programação. As principais abstrações utilizadas são as classes `Mat` e `Point_`, que definem a imagem em si e os *pixels* da imagem, respectivamente. A classe `Point_` permite a indexação de cada *pixel* por sua coordenadas cartesianas, tornando simples a edição de valores individuais de cada um. Já a classe `Mat` possuí vários métodos que facilitam a implementação de algoritmos de análise e manipulação de imagens, como a obtenção de de colunas e linhas específicas pelos métodos `Mat::col` e `Mat::row`.

Por ser um *software* utilizado mundialmente em pesquisa e desenvolvimento, ele ainda apresenta a vantagem de já ter integrado diversos algoritmos. Métodos como detecção de borda, cálculo de momentos de imagem, calibração de câmera e *Machine Learning* já sáo embarcados na biblioteca.

Nesse projeto, foram utilizados métodos simples de reconhecimento de objetos e formas. O funcionamento desses métodos será explicado em maior detalhe no próximo capítulo. Por último, é válido mencionar que, por se tratar de uma imagem obtida por meio de uma câmera simulada, não há erros relacionados a distorções da imagem pela lente e portanto não há a necessidade de implementar correção para esse tipo de erro.

Capítulo 3

Métodos

Para esse projeto, o algoritmo de *Machine Vision* foi dividido em duas partes. A primeira parte consiste em identificar as bordas de contorno dos objetos, eliminando ruídos no processo e encontrando os furos. Na segunda parte é necessário corrigir a perspectiva da câmera (homografia), a fim de obter a posição correta do furo para o posicionamento do manipulador.

3.1 Detecção de borda Canny

A deteção de borda utilizada pelo OpenCV foi originalmente desenvolvida por John Canny [5]. Ela é um processo constituído por 4 passos:

1. Aplicar um filtro gaussiano na imagem para diminuir ruído.

2. Encontrar o gradiente de intensidade da imagem. Para isso é utilizado um operador de detecção de borda, que retorna a primeira derivativa nas direções verticais e horizontais. Esse passo é subdividido em duas

partes: Na primeira são aplicadas um par de máscaras de convolução nas direções x e y.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

A segunda parte consiste em encontrar o módulo e a direção do gradiente de intensidade. A direção do gradiente é depois aproximada para uma de 4 opções (0, 45, 90 ou 135 graus).

$$G = \sqrt{(G_x)^2 + (G_y)^2} \tag{3.1}$$

$$\theta = arctan(\frac{G_y}{G_x}) \tag{3.2}$$

3. Aplica-se supressão não máxima nos pontos, removendo-se *pixels* que não são considerados parte da borda.

4. Seleciona-se os *pixels* da borda através de dois *thresholds*, um superior e outro inferior, da seguinte maneira:

   (a) Os *pixels* valores de gradientes acima do *threshold* superior são aceitos

(b) Os *pixels* valores de gradientes inferiores ao *threshold* inferior são descartados

(c) Os de valores intermediários são aceitos caso houver um *pixel* adjacente com o gradiente ultrapassando o *threshold* superior

3.2 Homografia

No modelo ideal de uma câmera, a relação entre duas imagens é calculada através de homografia ou transformação projetiva [6]. Ela é uma transformação linear representada por uma matrix não singular 3x3, quer elaciona os pontos de duas imagens.

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_1 1 & h_1 2 & h_1 3 \\ h_2 1 & h_2 2 & h_2 3 \\ h_3 1 & h_3 2 & h_3 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Isso é calculado de forma que o erro de projeção, definido pela fórmula 3.3, seja mínimo.

$$\sum_i (x'_i - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}})^2 + (y'_i - \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}})^2 \qquad (3.3)$$

Ao selecionar os mesmos pontos de um objeto em duas imagens que possuam perspectivas diferentes, é possível calcular a matriz transformação entre essas duas imagens. Assim, aplicando-se essa transformação à primeira imagem, obtêm-se primeira imagem sobre a perspectiva da segunda imagem,

conforme mostra a figura 3.1.

Figura 3.1: Exemplo de correção de perspectiva por homografia
Fonte: Harley e Zisserman [6]

Capítulo 4

Resultados e Discussões

4.1 Organização do Projeto

No início do projeto definiram-se algumas simplificações do modelo em relação ao que seria um ambiente real. Entre elas, destaca-se:

1. A surpefície de encaixe de parafusos, que pode ter uma modelagem complexa em uma aplicação real, foi substituída por uma placa plana com furos circulares. Essa, por sua vez, é facilmente modelável pelo *software*, simplificando a simulação e evitando possíveis erros de colisão que podem ser encontrados em uma superfície mais complexa.

2. Para a detecção dos furos, foi utilizada apenas uma câmera. Dessa forma, não é possível realizar a triangulação e encontrar a profundidade dos objetos na cena. Isso foi solucionado adicionando-se duas restrições: manter a distância relativa da base móvel e da placa constante e adicionar uma referência para a câmera, no caso, as linhas

verticais em verde na placa. Esse método de calibração poderia ser substituído pelo método mais comumente utilizado de calibração por uma matriz (normalmente um tabuleiro de xadrez) desenvolvido por [7].

3. Não houve modelagem dos parafusos nem do seu encaixe na placa. Ao invés disso, a montagem simulada foi simplificada pela chegada da ponta no furo, com precisão de até 5 mm.

Considerando essas simplificações, foi elaborada uma estrutura de comunicação dividida entre os *scripts* embarcados do V-REP e dois executáveis de controle compilados pelo ROS, denominados *omnipad* e *sawyer_robot*, que controlam a base móvel e o braço mecânico, respectivamente. Além disso, o *sawyer_robot* também toma decisões de avanço ou parada e realiza a análise de imagem enquanto o V-REP realiza os cálculos necessários para a cinemática inversa. A Figura 4.1 demonstra a composição de comunicação entre os diversos nós.

A composição da cena para simulação do V-REP foi organizada com dois modelos *built-in*, onde o braço mecânico *sawyer*, foi montado sobre a base móvel *Omnipad*. A parede foi desenhada no *software blender* e importada, montando a cena conforme mostra a Figura 4.2.

Com o cenário e a comunicação entre os programas elaborados, estabeleceu-se a lógica de controle dos programas, conforme demonstrado no fluxograma da figura 4.3

Informações enviadas:

1.
- Comando para inicializar a cinemática inversa
- Posição da localização do furo

2.
- Tempo de simulação
- Imagem da câmera
- Aviso do término da cinemática inversa

3.
- Aviso para a base andar ou parar

4.
- Velocidade dos motores

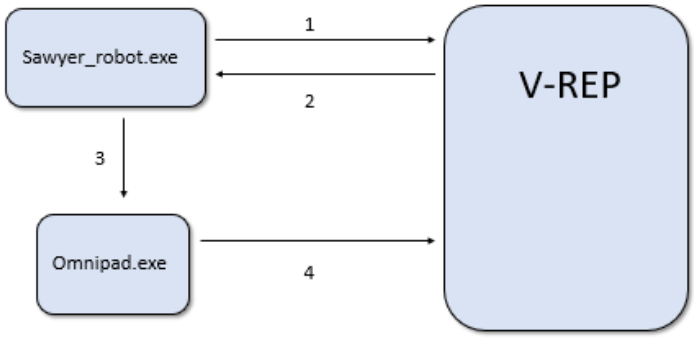

Figura 4.1: Comunicação entre os nós

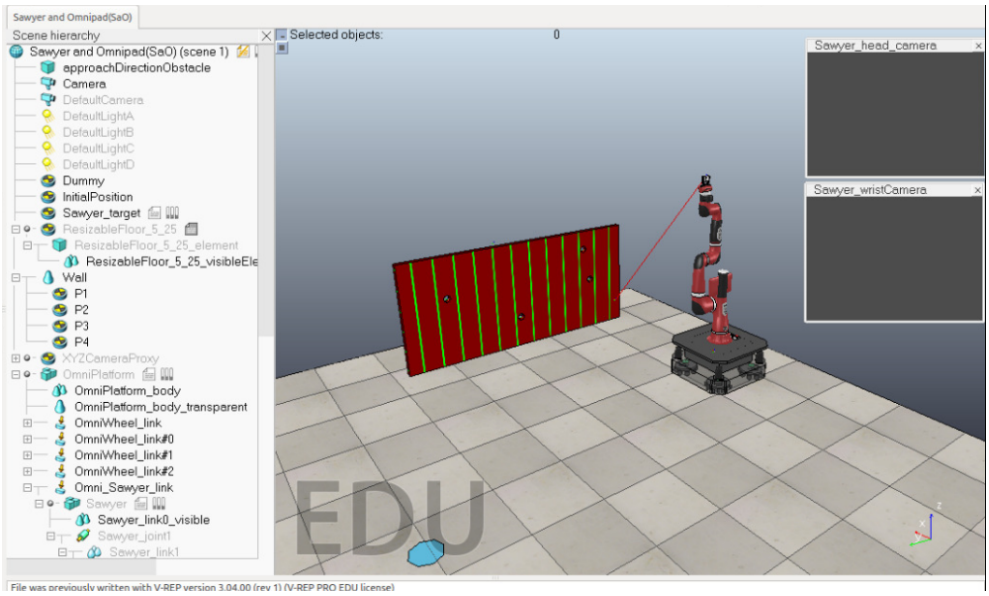

Figura 4.2: Organização da cena

4.2 Algoritmo de visão

O algoritmo pode ser divido em duas partes. A primeira parte busca os furos de parafuso que devem ser colocados enquanto a segunda parte procura a

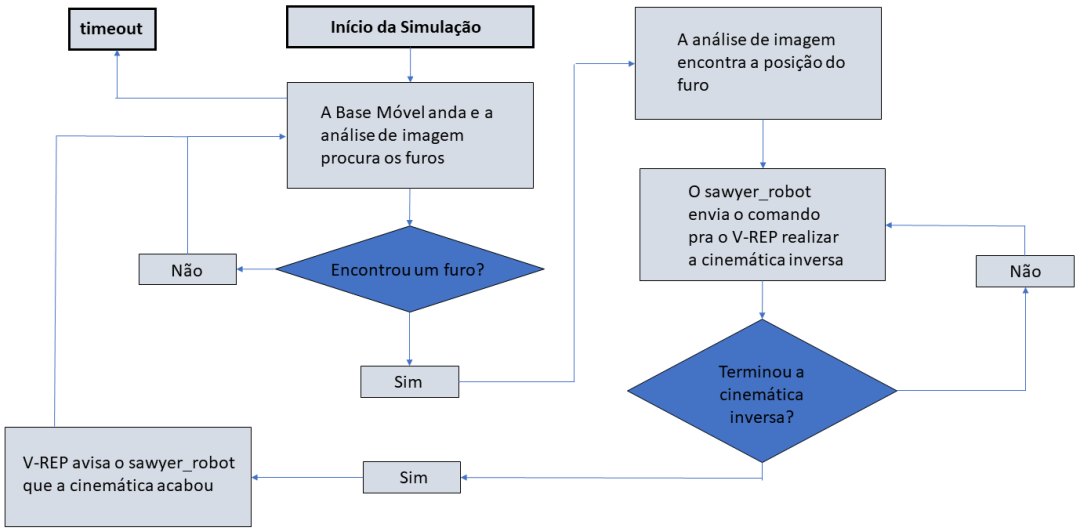

Figura 4.3: Fluxograma da simulação

altura relativa deste à base da placa.

Durante a primeira parte, o algoritmo segue os passos mostrados pela Figura 4.4.

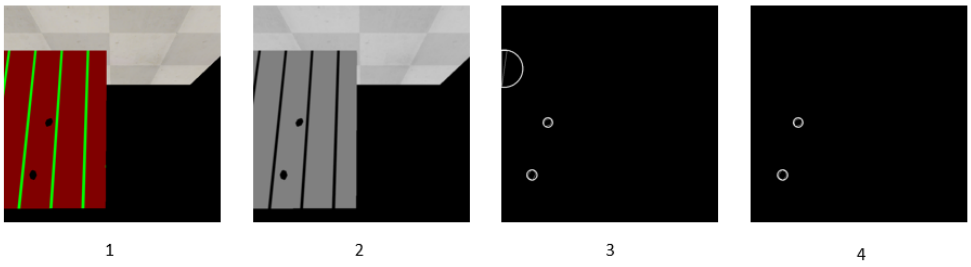

Figura 4.4: Primeira parte do algoritmo de visão

1. Imagem obtida pela câmera

2. Conversão para tons de cinza

3. Aplica-se *threshold* e o detector de borda

4. Retira-se ruído da imagem

Ja durante a segunda parte, os mesmos passos são realizados para os retângulos de retificação, obtidos através das linhas de referência na placa. Encontra-se a posição dos vértices do retângulo central, e estes servem de base para a homografia. Na imagem modificada, calcula-se a altura relativa da posição do furo em relação à base, e esta é enviada para que o V-REP possa calcular a cinemática inversa. As transformações na imagem são mostrados na Figura 4.5.

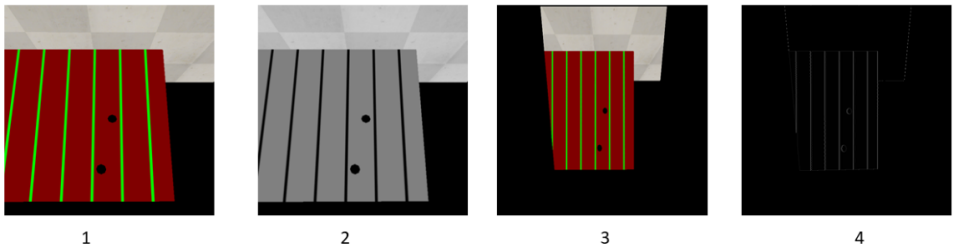

Figura 4.5: Segunda parte do algoritmo de visão

Durante as simulações obteve-se os gráficos de velocidade e aceleração da ponta do braço mecânico, as Figuras 4.6 e 4.7 apresentam os os maiores valores obtidos durante os testes. Todas as juntas estavam no modo Torque, o que significa que assim que o movimento da cinemática se iniciava, elas recebiam imediatamente o toque máximo disponível, gerando valores

elevados de Velocidade e Aceleração, com picos de aproximadamente $3m/s$ e $22m/s^2$.

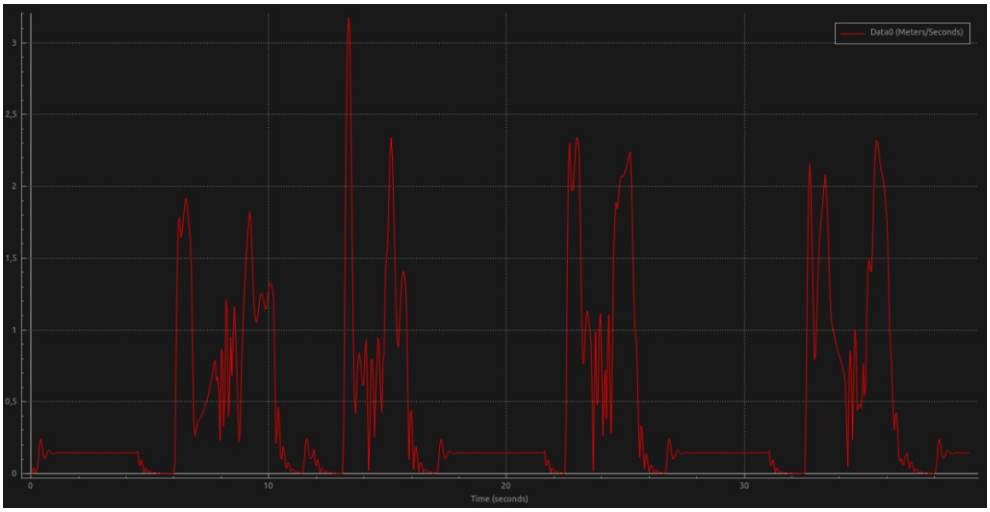

Figura 4.6: Perfil de velocidade

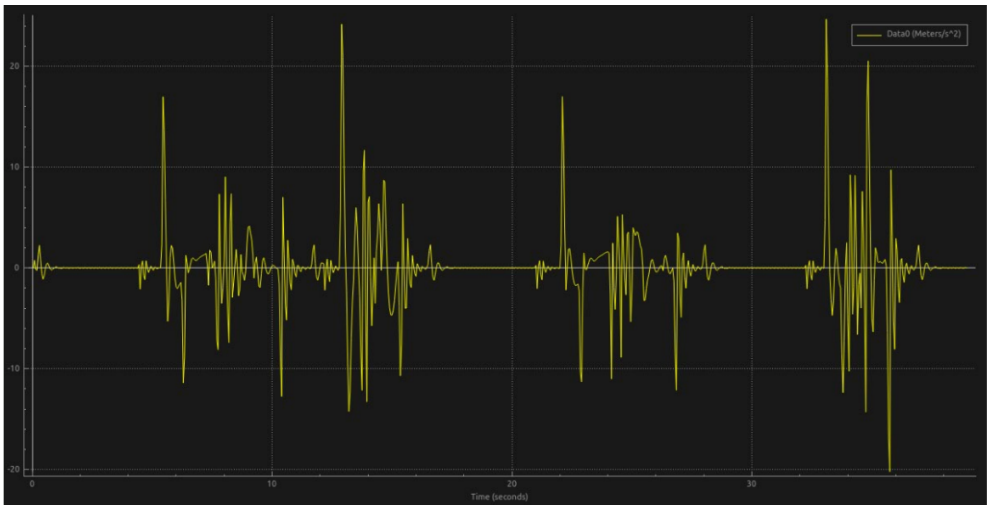

Figura 4.7: Perfil de aceleração

Para a verificação desses valores, tentou-se comparar os dados simula-

dos com os dados do robô físico, mas a *Rethink Robotics* não forneceu os dados necessários após contato. Portanto, comparou-se os resultados com um manipulador de porte semelhante, o KUKA KR 16-2; que possuí o alcance máximo de 1610 mm, enquanto o Sawyer possuí alcance de 1026mm; durante um perfil de alta velocidade. Os perfis de velocidade e aceleração são mostrados na figura 4.8.

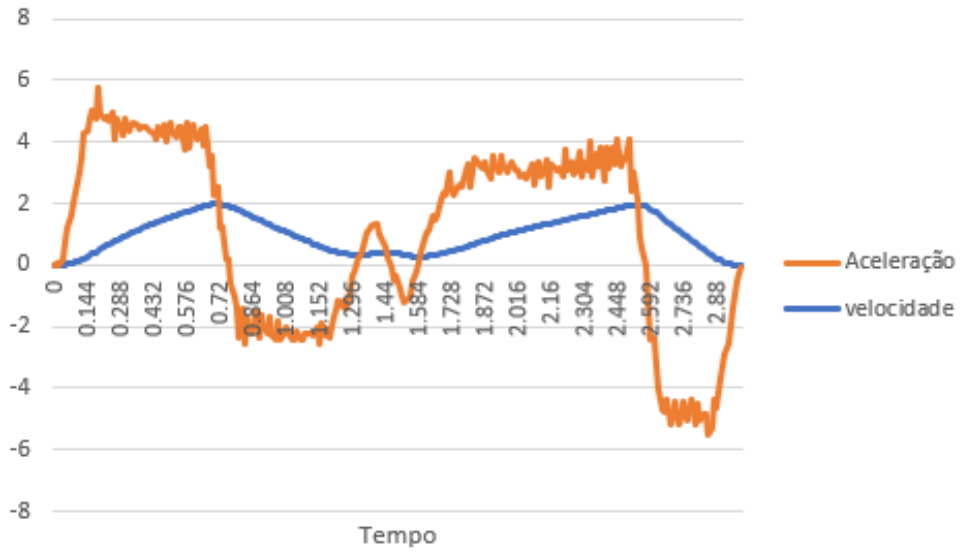

Figura 4.8: Perfil de aceleração

Nota-se que há uma grande diferença nos valores obtidos pelos dois robôs. Como o kuka possuí um alcance maior, é esperado que as velocidades e as acelerações dele fossem maiores, mas o inverso ocorre. Essa diferença pode ser explicada em principalmente pelo modo de controle da junta, que não é compatível com aplicações reais, e com a trajetória do TCP, que foi mais simples no caso do kuka.

Durante o desenvolvimento do projeto, houveram dificuldades em relação

ao uso do *software* V-REP, em especial em relação à robustez e à documentação online disponível. Entre os principais problemas encontrados, detaca-se que quando uma simulação acabava devido a um erro, como o de utilizar um tipo de junta errada, houve instâncias em que as propriedades do modelo modificaram-se sozinhas e as informações erradas se mantinham para as próximas simulações. O caso mais comum foi o de ângulos iniciais das juntas mudarem, mas também houve casos de outras propriedades mudarem, comprometendo a estabilidade do modelo durante as próximas simulações. Em relação à documentação que encontra-se disponível *online*, a explicação sobre o funcionamento das APIs é insuficiente para o entendimento da mesma, com pouca explicação sobre as variáveis necessárias para o seu funcionamento correto. Apesar de ser demorado, a melhor maneira de compreender a linguagem de programação do *software* é através dos exemplos disponíveis ao se instalar o programa. Nesse mesmo tópico, é importante enfatizar que nem todos os parâmetros da simulação podem ser controlados através de linhas de código, limitando as possibilidades de programação.

Capítulo 5

Conclusão

O projeto demonstrou a possibilidade da utilização dessas ferramentas em conjunto para o desenvolvimento de pesquisas em manipulação móvel. O ROS e a biblioteca OpenCV são conhecidos pelo seu uso na área da robótica em diversas aplicações e funcionaram conforme o esperado durante o projeto. Já o V-REP apresentou problemas durante as simulações que podem dificultar o seu uso em projetos mais complexos. Caso tais desvantagens sejam corrigidas em versões futuras do *software*, a sua versatilidade e abrangência podem tornar o seu uso extremamente útil para testes preliminares não só de manipulação móvel, mas também de diversas áreas da robótica.

Apêndice A

Roteiro

Esse roteiro tem como objetivo auxiliar pessoas que tenham o interesse de utilizar o V-REP em conjunto com o ROS. Além de mostrar os passos necessários para o ajuste das configurações, ele também irá abordar os principais aspectos da simulação no V-REP, a fim de preservar características importantes para a engenharia como dados de velocidade e aceleração realistas.

Para a instalação completa do ROS, em um terminal do ubuntu, é necessário entrar com os seguintes comandos:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
-sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Esse comando habilita o seu sistema operacional a aceitar o *software* fornecido pela *packages.ros.org*.

Em seguida, é necessário entrar com a *keyserver*, utilizada para segurança do servidor, com o comando:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
--recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Antes da instalação do *software* em si, é sempre recomendável no linux atualizar a lista de informações sobre os pacotes do sistema, isso é feito com o comando:

```
sudo apt-get update
```

Enfim, para a instalação completa do *software*:

```
sudo apt-get install ros-distro-desktop-full
```

Onde distro é a versão do ROS a ser instalada e depende da versão do ubuntu. Se o ubuntu é 16, então o ROS será kinect; se é o 14, então será o indigo.

Além disso, é conveniente adicionar os comandos ROS ao terminal. Isso é feito editando-se o programa que inicializa os terminais do linux:

```
echo "source /opt/ros/kinetic/setup.bash>>  /.bashrc
source  /.bashrc
```

Nesse ponto da instalação é importante mencionar sobre o modo de compilação do ROS, já que para o V-REP ele funciona de forma diferente. Por conveniência, todos os comandos necessários para compilação da sua *workspace* são resumidos em um só, o `catkin_make`. Portanto, para uma utilização padrão do ROS, o processo consiste em criar uma pasta que será o workspace, por convenção chamada de *catkin_ws*, criação de uma pasta source dentro, inicialização e compilação da workspace.

Esses passos são implementados com os seguintes comandos:

Criação da pasta *catkin_ws* e da pasta *src* dentro:

```
mkdir -p  /catkin_ws/src
```

Mudar o diretório para a pasta *src*:

```
cd  /catkin_ws/src
```

Comando para a criação do *workspace*:

```
catkin_init_workspace
```

Mudar o diretório para a pasta *catkin_ws*:

```
cd  /catkin_ws/
```

Compilar o *workspace*:

```
catkin_make
```

Porém, o V-REP não utiliza o *catkin_make*, mas sim o *catkin_tools*, que pode ser instalado através dos comandos:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main» /etc/apt/sources.list.d/ros-latest.list'
```

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

```
sudo apt-get update
sudo apt-get install python-catkin-tools
```

Feito isso, basta seguir o padrão de criação de uma workspace, substituindo o `catkin_make` pelo `catkin build`:

```
mkdir -p  /catkin_ws/src

cd  /catkin_ws/src

catkin_init_workspace

cd  /catkin_ws/

catkin build
```

Já para a instalação do V-REP, basta realizar o *download* na página oficial da *Coppelia Robotics*. O programa já vem pronto para ser utilizado.

Agora é necessário configurar o V-REP para ser considerado como um nó pelo ROS. O primeiro passo é copiar os arquivos *v_repExtRosInterface*, *ros_bubble_rob2*, *vrep_skeleton_msg_and_srv* e *vrep_plugin_skeleton* para a pasta *catkin_ws/src*. Elas estão dentro da pasta *programming/ros_packages* no diretório do V-REP. Em seguida compila-se o *workspace* do ROS com o comando `catkin build`, para que ele passe a reconhecer o V-REP.

Para testar a conexão, é necessário iniciar o *master* em um terminal com o comando `roscore`. Em um segundo terminal, no diretório do V-REP, utilizar o comando `./vrep.sh`. Esse comando inicializa um *script* que contém todos os comandos necessários para a inicialização do *software*. A

cena *controlTypeExampls.ttt* testa todos os métodos de controle possíveis para o programa.

Com a conexão estabelecida, a elaboração de projetos que utilizam os *softwares* em conjunto já é possível. Porém há algumas características sobre a simulação que devem ser levados em consideração, pois eles influenciam a qualidade dos resultados.

O código do V-REP é organizado em *scripts*, divididos em *main Script* e *child scripts*. A *main script* é a parte do código que coordena a simulação como um todo e recomenda-se não modifica-lo. Já as *child scripts* coordenam os modelos e são separados em *threaded* e *non-thread*.

Threaded child scripts irão funcionar dentro de *threads* que se revezam a cada 1-2 milisegundos e reiniciam o processamento a cada novo passo da simulação. Seu objetivo é de imitar o comportamento de co-rotinas sem utilizá-las. Elas possuem desvantagens em relação as *non-threaded child scripts*, como maior uso de recursos computacionais e menor eficiência.

Já as *Non-Threaded child scripts* são chamadas pelo *main Script* duas vezes por passo de simulação. São compostas por 4 funções: no início da simulação uma função declara as variáveis; durante a simulação há uma função que controla os sensores e outra os atuadores, e por último uma que encerra o código. Caso as *non threaded child scripts* não retornem o controle para o *main script*, elas podem travar toda a simulação.

As juntas são divididas em 4 tipos: Revolução, prismática, parafuso e esférica. Além disso, cada junta possuí 4 possíveis modos de funcionamento: passiva, cinemática inversa, dependente e torque, com a possibilidade de estabelecer um controle híbrido entre o modo torque e os outros três. O

modo torque ainda suporta 3 modelos de *loop* de controle: customizado, PID e mola-amortecedor.

Para obter o controle preciso das juntas como o modo de torque, são recomendadas duas opções, a primeira é controlar a junta através de uma aplicação externa sem o *loop* de controle, e a segunda é controlar a junta pelo V-REP com o *loop*. Nesse projeto utilisou-se a segunda opção. Um outro ponto importante é o tamanho do *time step* da simulação. Embora o padrão do V-REP seja 50 ms, o motor de física faz cálculos para cada 5 ms, o que pode gerar divergência de valores realistas ao longo da simulação. Portanto é recomendável utilizar valores inferiores.Para alguns modelos de robô, principalmente modelos complexos, é recomendável utilizar valores ainda menores como 1 ms, a fim de que os resultados da simulação sejam os mais próximos da realidade.

A seguir, demonstra-se o código necessário para estabelecer uma comunicação simples entre os dois softwares:

Na pasta `catkin_ws/src` cria-se a pasta que irá conter o projeto, com o nome de exemplo. Para cada projeto, são necessários no mínimo três arquivos:

1. CmakeLists.txt
2. package.xml
3. scr/exemplo.cpp

O CmakeLists.txt e package.xml incluem dependências e especificações necessárias para a compilação de projetos do ROS. O package.xml contém

informações de descrição sobre o projeto em si e também informações de dependências enquanto o CMakeLists.txt também contém dados sobre a versão do *cmake* e o endereço de onde o arquivo binário gerado pela compilação será criado, no caso, ele já é gerado diretamente no diretório do V-REP.

Arquivo package.xml:

```
<package>
  <name>exemplo</name>
  <version>0.0.0</version>
  <description> Exemplo comunicacao V-REP e ros </description>
  <maintainer email="exemplo@email.com"</maintainer>
  <license>TODO</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>geometry_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>geometry_msgs</run_depend>
</package>
```

Arquivo CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.3) #versão do cmake
project(exemplo) # nome do projeto

#nome das depdendências do projeto
find_package(catkin REQUIRED)
```

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  geometry_msgs
)

include_directories(${catkin_INCLUDE_DIRS})
#adiciona o endereço do executável
set(EXECUTABLE_OUTPUT_PATH /home/user/Documents/vrep)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

#comando que cria o executável, com o formato
#add_executable(NOME_EXECUTAVEL ARQUIVO_FONTE)
add_executable(exemplo src/exemplo.cpp)
target_link_libraries(exemplo ${catkin_LIBRARIES})
add_dependencies(exemplo ${catkin_EXPORTED_TARGETS})
```

Os próximos dois códigos estabelecem a comunicação. São criados dois tópicos, um do V-REP para o binário que o catkin build gera e o outro no sentido inverso. O V-REP manda a informação de tempo de simulação para o exemplo e esse retorna a informação de volta. Esse modelo foi baseado no projeto ros_bubble_rob2.

Arquivo exemplo.cpp:

```
#include <stdio.h>
#include <stdlib.h>
#include <ros/ros.h>
#include "std_msgs/Float32.h"
```

```
struct timeval tv;// informação de tempo do sistema, será usada para adicionar uma componente randômica ao
unsigned int currentTime_updatedByTopicSubscriber=0;//inicialização do tempo do sistema
float simulationTime=0.0; // tempo de simulção que será fornecido pelo V-REP

// Função de callback do tempo de simulação
void simulationTimeCallback(const std_msgs::Float32& simTime){
   simulationTime=simTime.data;
}

int main(int argc,char* argv[])
{
//Nomes Tópicos
    std::string RosToVrep;
    std::string simulationTimeTopic;

    if (argc>=2)
    {
     // Ao chamar o executável, o V-REP manda nomes personalizados
    //isso é feito para permitir várias instâncias do mesmo modelo funcionando juntas
      RosToVrep=argv[1];
      simulationTimeTopic=argv[2];
      RosToVrep="/"+RosToVrep;
      simulationTimeTopic="/"+simulationTimeTopic;
    }
    else
    {
        printf("Não há parametros suficientes!\n");
        sleep(5000);
        return 0;
```

```
}

// Criando o nome do nó com componente randômico
  int _argc = 0;
  char** _argv = NULL;
  if (gettimeofday(&tv,NULL)==0) //adicionar variavel randomica para o nome
  currentTime_updatedByTopicSubscriber=(tv.tv_sec*1000+tv.tv_usec/1000)&0x00ffffff;
  std::string nodeName("exemplo");
  std::string randId(boost::lexical_cast<std::string>(currentTime_updatedByTopicSubscriber+int(999999.
  nodeName+=randId;

   ros::init(_argc,_argv,nodeName.c_str()); //iniciar o no
   if(!ros::master::check())  return(0);
   ros::NodeHandle n;
   printf("O exemplo está com o nome  %s\n",nodeName.c_str());

  //Declaração Assinante, que indica a função de callback
  ros::Subscriber subSimulationTime=n.subscribe(simulationTimeTopic.c_str(),1,simulationTimeCallback);
  //Declaração Editor
  ros::Publisher RosToVrepPub=n.advertise<std_msgs::Float32>(RosToVrep.c_str(),1);

  //Atualização do tempo pelo sistema
  float driveBackStartTime=-99.0f;
  unsigned int currentTime;
  if (gettimeofday(&tv,NULL)==0)
  {
      currentTime_updatedByTopicSubscriber=tv.tv_sec;
      currentTime=currentTime_updatedByTopicSubscriber;
  }
```

```
      //Inicio Loop
      while (ros::ok())
      {
         //break se o programa para de mandar informação
         if (gettimeofday(&tv,NULL)==0)
        {
            currentTime=tv.tv_sec;
            if (currentTime-currentTime_updatedByTopicSubscriber>1000)
            break;
        }

        std_msgs::Float32 d;
        d.data=simulationTime;
        RosToVrepPub.publish(d);

 //Processa as mensagens
        ros::spinOnce();

        //Tempo em pausa
        usleep(10000);
    }
        ros::shutdown();
        printf("Terminou o exemplo!\n");
        return(0);
}
```

Script do V-REP:

```
if (sim_call_type==sim_childscriptcall_initialization) then
```

```
handle = simGetObjectHandle('demonstracao')


------Testando o Ros Interface-----
    moduleName=0
    moduleVersion=0
    index=0
    pluginNotFound=true
    while moduleName do
        moduleName,moduleVersion=simGetModuleName(index)
        if (moduleName=='RosInterface') then
            pluginNotFound=false
        end
        index=index+1
    end
----------------------------------



 if (not pluginNotFound) then
       local sysTime = simGetSystemTimeInMs(-1)
       local simulationTimeTopicName='simTime'..sysTime
       local RosToVrepTopicName = 'RosToVrep'..sysTime
       simTimePub=simExtRosInterface_advertise('/'..simulationTimeTopicName,'std_msgs/Float32')
       RosToVrepSub=simExtRosInterface_subscribe('/'..RosToVrepTopicName,'std_msgs/Float32','RosToVrep_cal
       result=simLaunchExecutable('exemplo',RosToVrepTopicName.." "..simulationTimeTopicName,0)
    end


    con=simAuxiliaryConsoleOpen('console',100,0101)



end
```

```
if (sim_call_type==sim_childscriptcall_actuation) then
    local sT=simGetSimulationTime()
    simExtRosInterface_publish(simTimePub,{data=sT})
end


if (sim_call_type==sim_childscriptcall_sensing) then
end


if (sim_call_type==sim_childscriptcall_cleanup) then
end


function RosToVrep_callback(msg)
    if msg.data then
     simAuxiliaryConsolePrint(con, msg.data.."\n")
    end
end
```

Referências Bibliográficas

[1] M. et al; Hvilshøj. Autonomous industrial mobile manipulation (aimm): past, present and future. *Industrial Robot: An International Journal*, 39(2):120–135, 2012.

[2] Comitê Técnico de Manipulação Móvel da IEEE. Disponível em: http://www.ieee-ras.org/mobile-manipulation, Acesso em 07 de abril 2018.

[3] Darpa Challenge. Disponível em: https://www.darpa.mil/program/darpa-robotics-challenge, Acesso em 07 de abril 2018.

[4] N. et al; Karlsson. The vslam algorithm for robust localization and mapping. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 24–29, 2005.

[5] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[6] Andrew Zisserman Richard Hartley. *Multiple view geometry in computer vision*. Cambridge University Press, Reading, Massachusetts, 2004.

[7] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, 2000.