UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
FACULDADE DE CIÊNCIAS EXATAS
CURSO DE SISTEMAS DE INFORMAÇÃO

DESENVOLVIMENTO DE UMA APLICAÇÃO PARA OPERAÇÃO USANDO O ROBOT BAXTER

Tommaso Bellone

Diamantina

2018

UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI
FACULDADE DE CIÊNCIAS EXATAS

DESENVOLVIMENTO DE UMA APLICAÇÃO PARA OPERAÇÃO USANDO O ROBOT BAXTER

Tommaso Bellone

Orientador:

Rafael Santin

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação, como parte dos requisitos exigidos para a conclusão do curso.

Diamantina

2018

DESENVOLVIMENTO DE UMA APLICAÇÃO PARA OPERAÇÃO USANDO O ROBOT BAXTER

Tommaso Bellone

Orientador:

Rafael Santin

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação, como parte dos requisitos exigidos para a conclusão do curso.

APROVADO em 21 / 01 / 2019.

Prof. Me. Luis Henrique Silva Rodrigues – UFVJM

Prof. Me. Marcelo Ferreira Rego – UFVJM

Prof. Me. Rafael Santin – UFVJM

Aos meus filhos Elisa e Tommaso Júnior,
preciosidades da minha existência

AGRADECIMENTO

Um particular agradecimento para o Professor Rafael Santin pela paciência e atenção durante a orientação deste trabalho e pela grande oportunidade de aprendizado. Agradeço também aos professores, técnicos e funcionários do Departamento de Computação que me acompanharam ao longo desta prazerosa jornada.

Na vida, ao contrário do xadrez, o jogo continua após o xeque-mate.

-Isaac Asimov-

RESUMO

Um robô é um dispositivo eletromecânico ou biomecânico capaz de realizar trabalhos de maneira autônoma ou pré-programada. A utilização de robôs industriais nas linhas de produção está crescendo exponencialmente. Eles são recomendados para atividades repetitivas e que tenham alguma condição de risco ou ambiente insalubre. O manipulador é um robô multifuncional reprogramável, projetado para mover materiais, peças, ferramentas ou dispositivos através de vários movimentos programados para a execução de várias tarefas.

Nesse sentido, o objetivo do presente trabalho é o estudo das principais ferramentas da atualidade que permitam operar um manipulador simulado. Para isso, elaboramos uma aplicação para intergir com objetos virtuais usando o simulador do manipulador.

As ferramentas utilizadas foram:

- ROS, um framework que cria um ambiente que auxilia na escrita de software para robôs;
- Gazebo, um simulador robótico dinâmico em um ambiente virtual capaz de simular grupos de robôs, sensores e objetos em um modelo virtual;
- Software do robô Baxter que foi usado como base do projeto;
- MoveIt!, um software para a manipulação móvel, incorporando os últimos avanços em planejamento de caminhos sem colisões.

Visando possibilidades futuras da utilização do trabalho, foram gerados documentos explicativos para execução das ferramentas e da aplicação produzida.

Palavras-chave: ROS, Manipulação de Robôs, Cinemática Reversa, Gazebo.

ABSTRACT

A robot is an electromechanical or biomechanical device capable of performing tasks in an autonomous or pre-programmed manner. The use of industrial robots in production lines is growing exponentially: they are recommended for repetitive or hazardous activities, or in unhealthy environments. The manipulator is a reprogrammable multifunctional robot, designed to move materials, parts, tools or devices in order to perform tasks.

The aim of the present work is to study the main current tools that allow operating a simulated manipulator. To achieve this, we have developed an application to interact with virtual objects using the manipulator simulator.

The tools used were:

- ROS, a framework that creates an assisted environment to write software for robots.
- Gazebo, a virtual environment dynamic robotic simulator, capable of representing a simulated model of groups of robots, sensors and objects.
- Baxter's robot software, used as a basis for the project.
- MoveIt! software for mobile manipulation, that incorporates the latest advances in collision-free path planning.

Keywords: Robotic Operating System, Robot Manipulation, Reverse Kinematics, Gazebo.

LISTA DE FIGURAS

LISTA DE SIGLAS

ACM	Allowed Collision Matrix
DART	Dynamic Animation and Robotics
DOF	Degree Of Freedom
FCL	Flexible Collision Library
GUI	Graphics User Interface
IA	Inteligência Artificial
KDL	Kinematic and Dynamic Solvers
ODE	Open Dynamics Engine
OGRE	Object-Oriented Graphics Rendering Engine
OMPL	Open Motion Planning Library
ROS	Robot Operating System
SRDF	Semantic Robot Description Format
URDF	Unified Robot Description Format

SUMÁRIO

1 - INTRODUÇÃO.

A robótica é a disciplina que estuda e desenvolve métodos que permitem a um robô executar atividades similares aquelas de um ser humano.

Um robô é um dispositivo, ou grupo de dispositivos, eletromecânicos ou biomecânicos capazes de realizar trabalhos de maneira autônoma ou pré-programada. Os robôs são comumente utilizados na realização de tarefas em locais mal iluminados ou perigosos para os seres humanos. Os robôs industriais utilizados nas linhas de produção possuem precisão, excelente agilidade e repetibilidade. São recomendados para atividades repetitivas e que tenham alguma condição de risco ou ambiente insalubre. Outras aplicações são: exploração subaquática e espacial, cirurgias, mineração, busca e resgate, manuseio de lixo tóxico e localização de minas terrestres. Atualmente os robôs também aparecem nas áreas do entretenimento e tarefas caseiras (ROS, 2016). A robótica envolve diferentes áreas de pesquisa como linguística, biologia, fisiologia, psicologia, informática, matemática e mecânica. Quando, em 1950, foi publicada a antologia de Isaac Asimov, "Eu robô", somente os futurólogos mais ousados da época poderiam ter imaginado que a curva de crescimento tecnológico teria sido projetada setenta anos depois em um cenário com as perspectivas atuais. Hoje, a robótica está entrando em nossas vidas domésticas e profissionais (USP, 2014). A robótica é o ramo da engenharia mecatrônica em que confluem os estudos de muitas disciplinas da natureza (ROS, 2014) humanística, como a fisiologia, a linguística e a psicologia e da natureza científica como a automação, a biologia, a eletrônica, a física, a informática, a matemática e a mecânica (ROS, 2013).

O termo "robot"foi criado pelo escritor checo Karel Capek em 1920 para um drama teatral. No seu idioma "robota"significava trabalho forçado; na verdade, as primeiras máquinas robô foram utilizadas nos ciclos produtivos de fabricação (FAIRCHILD; HARMAN, 2016). O desenvolvimento tecnológico levou a novas utilizações dos robot, (GAZEBO, 2017) no ambiente doméstico e no ambiente lúdico. Mas o grande avanço da robótica se deve a indústria aerospacial, a seguir alguns exemplos(Figura 1.1)

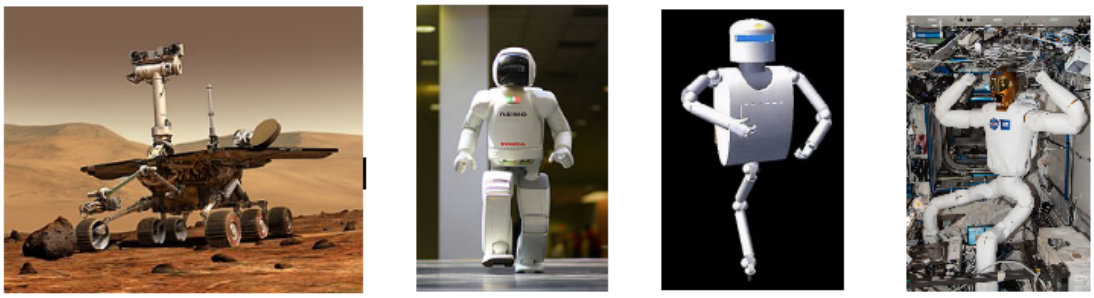


Figura 1.1: Robôs Spirit, Asimo, Robonaut e Robonaut 2 Fontes: ROVER2018, HONDA2011 e NASA2014

Spirit (MER-A) foi um veículo de exploração espacial não tripulado, cuja missão era estudar o planeta Marte, permanecendo ativo de 2004 a 2010. Foi um dos veículos projetados pela NASA para o Programa Mars Exploration Rovers. Pousou com sucesso em Marte em 3 de janeiro de 2004, três semanas antes do outro veículo, Opportunity (MER-B). Seu nome foi escolhido em uma competição estudantil promovida pela NASA. O robô ficou preso durante o seu trajeto em 2009 e perdeu contato com o Centro de Controle da missão em 22 de março de 2010 (ROVER, 2018).

ASIMO é um robô de 1 metro de altura e 52 quilogramas produzido pela Honda. Seu nome, curiosamente, não é uma referência ao escritor russo de ficção científica Isaac Asimov. Em japonês, ASIMO é pronunciado ashimo, que significa algo como "também com pernas". A sigla ASIMO em Inglês significa Advanced Step in Innovative Mobility, ou seja, Desenvolvimento Avançado em inovações para Mobilidade. Ele pode andar em superfícies irregulares, virar-se, pegar em coisas e reconhecer pessoas através das suas câmeras que funcionam como olhos (HONDA, 2011).

O Robonaut (de primeira geração foi projetado pelo Ramo de Tecnologia de Robot Systems no Centro Espacial Johnson da NASA em um esforço colaborativo com o DARPA (Defense Advanced Research Projects Agency).

Robonaut 2 foi revelado ao público em Fevereiro de 2010. R2 é quatro vezes mais rápido do que R1, mais compacto, mais qualificado e inclui uma maior faixa de detecção. Pode mover os braços até 2 m / s, pode levantar até 40 lb e as mãos têm a força para agarrar aproximadamente 5 libras por dedo. Existem também 350 sensores e 38 processadores PowerPC no robô (NASA, 2014).

1.1 Justificativa

Uma das aplicações mais populares para simuladores de robótica é a modelagem 3D e a renderização de um robô e seu ambiente. Este tipo de software de robótica simula um robô virtual, que é capaz de imitar os movimentos de um robô físico. Alguns simuladores de robótica usam um motor de física para gerar movimentos mais realistas para o robô.

O uso de um simulador de robótica para o desenvolvimento de um programa para controlar robôs é altamente recomendado, independentemente de o robô físico estar disponível ou não. O simulador permite que programas sejam convenientemente escritos e depurados off-line, e o software seja testado em robôs reais apenas depois de estarem maduros.

No especifico, serão apresentados alguns dos mais conhecidos robôs aprofundando, em particular, conceitos relativos a dois deles, o PR2 e o Baxter para depois estudar exemplos virtuais em ambientes de simuladores como Gazebo e MoveIt!.

1.2 Objetivos

Nesta seção serão apresentados os objetivos do presente trabalho.

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é compreender todo o ferramental necessário para utilizar o simulador Gazebo e propor o desenvolvimento de uma aplicação nesse ambiente.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são a pesquisa de elementos relativos a robótica, estudo do funcionamento do ROS (Robotic Operation System) para manipular, através software, os robôs e utilizar técnicas para manipular objetos transferido-os de uma ubicação para outras.

1.3 Organização

O presente trabalho é composto por cinco capítulos assim organizados:

- Capítulo 1: Introdução do trabalho;
- Capítulo 2: Referencial teórico compreendente manipuladores, cinemática e una breve descrição das ferramentas utilizadas;
- Capítulo 3: Metodologia utilizada;
- Capítulo 4: Apresentação dos resultados do trabalho aplicativo;
- Capítulo 5: Conclusões e possíveis sugestões para futuros trabalhos.

2 - REFERENCIAL TEÓRICO.

Este capítulo apresenta os conceitos teóricos e as ferramentas consideradas indispensáveis ao desenvolvimento e compreensão deste trabalho. São apresentadas questões relativas à Robótica, seja do ponto de vista histórico bem como do "estado da arte", a cinemática, o planejamento de caminhos e as ferramentas ROS e Gazebo.

2.1 Robótica - Manipuladores

Um manipulador é um dispositivo mecanicamente concebido para posicionar e orientar no espaço o seu órgão terminal: uma garra ou uma ferramenta. Pode ser, também definido como um conjunto de corpos ligados por juntas, formando cadeias cinemáticas que definem uma estrutura mecânica. No manipulador incluem-se os atuadores, que agem sobre a estrutura mecânica modificando a sua configuração, e a transmissão, que liga os atuadores à estrutura mecânica. Os termos manipulador e robô são muitas vezes usados com a mesma finalidade, embora, não esteja formalmente correto (LOPES, 2002). Um manipulador é uma cadeia cinética composta por elos (corpos da cadeia) e juntas, articulações entre os corpos que conectam os elos e permitem a realização de movimentos de um elo em relação ao elo anterior (TRONCO, 2004).

Os manipuladores de robô podem ser classificados por vários critérios, como sua fonte de energia, ou como as juntas são acionadas, sua geometria ou estrutura cinemática, sua área de aplicação pretendida ou seu método de controle. Essa classificação é útil principalmente para determinar qual robô é adequado para uma determinada tarefa. Por exemplo, um robô hidráulico não seria adequado para aplicações de manipulação de alimentos ou salas limpas (SPONG, 2006).

O espaço de trabalho é definido pelas características do manipulador e consiste no volume total percorrido pelo efetuador (órgão terminal ou ferramenta) dadas todas as possibilidades de execução de movimento do robô. Esse espaço é limitado pela geometria do manipulador, bem como pelas restrições físicas das juntas (BECKER, 2008). O espaço de trabalho de um manipulador pode ser representado como em figura 2.1.

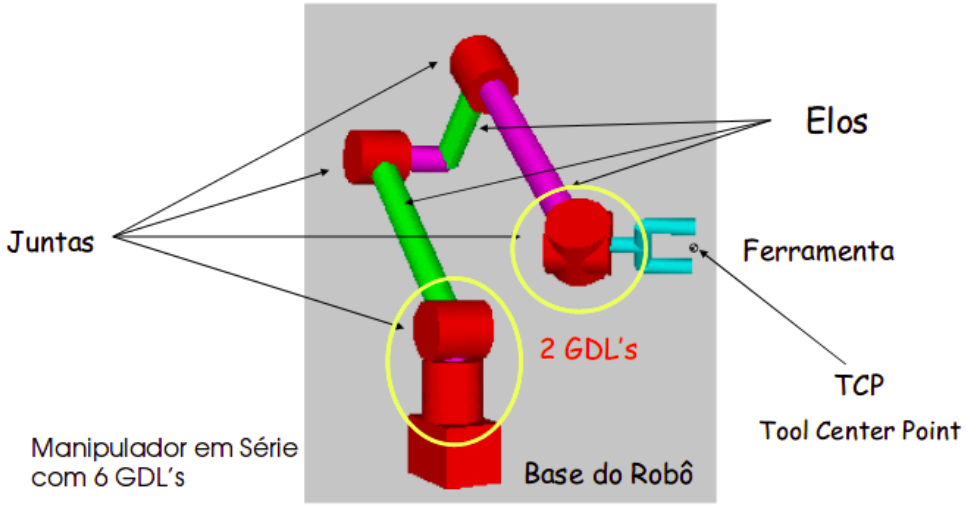


Figura 2.1: Espaço de trabalho de um manipulador Fonte: BACKER2008

2.2 Cinemática

A cinemática de um robô manipulador é o estudo da posição e da velocidade do seu efetuador e dos seus ligamentos. Quando se menciona posição, está se referindo tanto à posição propriamente dita, como à orientação e quando se fala em velocidade, considera-se tanto a velocidade linear como angular. Pode-se distinguir dois tipos de cinemática, a cinemática direta e a inversa. Na cinemática direta deseja-se obter a posição e velocidade do efetuador, para uma dada posição das articulações. A cinemática inversa é o oposto da cinemática direta, ou seja, são fornecidas a posição e a velocidade do efetuador e quer se obter as posições e velocidades correspondentes das articulações (CABRAL, 2012). A cinemática descreve o movimento de um ponto ou corpo no espaço utilizando equações matemáticas. Na robótica, particularmente na teoria que explica o movimento dos braços de um robô, pode haver 1, 2 ou mais graus de liberdade que correspondem aproximadamente às articulações que estes braços podem ter (assim como um braço humano que pode mover o cotovelo, que é um grau de liberdade, mas também o pulso que é outra articulação ou grau de liberdade, etc.). Os robôs podem ter vários graus de liberdade, dependendo de como são projetados. A composição entre as juntas do braço do robô possibilitam aumentar a liberdade de movimento (número de movimento) do braço, porém torna-se o seu controle mais complexo. No final, no robô, o arranjo dos braços será

tal que sua extremidade, geralmente uma espécie de pinça, será colocada em um ponto no espaço descrito por um conjunto de coordenadas. x, y, z.

2.2.1 Cinemática direta

Se as posições das juntas forem conhecidas, a cinemática direta transforma um ponto do espaço da junta na posição e orientação correspondentes da extremidade final; esse cálculo é bastante simples e não requer complexidade específica. Na prática, são fixadas a posição e o ângulo das articulações e se calcula a posição do final.

2.2.2 Cinemática inversa

A cinemática inversa, dadas a posição e a orientação final, determina todos os pontos correspondentes no espaço da articulação. Este problema inverso pode não ter solução se a posição exigida estiver fora do espaço de trabalho alcançável pelo braço robótico, ou tiver uma multiplicidade finita ou até infinita (robôs cineticamente redundantes).

2.3 Planejamento de caminhos

O processo de planejamento de caminho é um problema fundamental na robótica. A capacidade de analisar o ambiente e definir a sequência de ações que levam um robô de uma localização inicial até uma localização final desejada, sem colidir com os obstáculos presentes no ambiente é um elemento fundamental requerido para a criação de sistemas robóticos autônomos que possam executar diversas funções (SILVEIRA, 2017). Para decidir qual o caminho seguir a fim de realizar uma tarefa faz-se necessário algum tipo de planejamento. Decidir baseado em critério de otimalidade (caminho mais curto, caminho mais seguro, caminho que forneça o menor consumo de energia, etc.) é tratado na robótica como o problema de planejamento de caminhos (SANTANA, 2007).

Para o planejamento de caminho utilizamos neste trabalho a biblioteca OMPL (Open Motion Planning Library), a biblioteca de planejamento de movimento aberto, que consiste em muitos algoritmos avançados de planejamento de movimentos baseados em amostragem. O OMPL em si não contém nenhum código relacionado a, por exemplo,

verificação de colisão ou visualização. Essa é uma escolha deliberada de design, de modo que o OMPL não esteja vinculado a um determinado verificador de colisão ou front-end de visualização. A biblioteca é projetada para ser facilmente integrada a sistemas que fornecem os componentes adicionais necessários. Por exemplo, o OMPL é integrado ao sistema operacional de robótica ROS e à biblioteca Moveit! que fornece funcionalidade de planejamento de movimento no ROS.

2.4 ROS

ROS (Robot Operation System), indica uma estrutura software de código aberto muito utilizada no campo da robótica. Pode ser definido como um framework que auxilia na escrita de software para robôs, constituído de uma coleção de ferramentas, bibliotecas e convenções, tendo o objetivo de simplificar a criação de programas para tarefas complexas e robustas em robôs independente da plataforma. A filosofia principal do projeto ROS é facilitar a comunicação entre diferentes plataformas. Desta forma, o ROS incentiva o compartilhamento de código on-line, facilitando a integração dos sistemas. O ROS foi desenvolvido em 2007 no Stanford Artificial Intelligence Laboratory (SAIL), pela Willow Garage que foi um laboratório de pesquisa em robótica e incubadora de tecnologia dedicada ao desenvolvimento de hardware e software de código aberto para aplicações de robótica. Desde então teve um forte crescimento, o que a levou a ser uma das plataformas mais amplamente utilizadas no campo da robótica. Isso foi possível pelo compromisso de muitos institutos de pesquisa que iniciaram e desenvolveram projetos em ROS publicando-os sem restrições on-line seguindo a filosofia do software open-source. Além disso, o ROS permite desenvolver as aplicações de robótica em simuladores 2D e 3D. Um exemplo disso é Gazebo (GAZEBO, 2017), um simulador robótico dinâmico em um ambiente virtual, capaz de simular grupos de robôs, sensores e objetos em um modelo virtual. Também pode simular o feedback do sensor e as interações físicas realísticas entre objetos. Essencialmente, o ROS é baseado na construção de uma estrutura de processos chamados nós. Cada um deles assume funções específicas e a rede de nós construída é coordenada por um nó principal chamado roscore. Os vários nós são colocados em comunicação com o tópico fornecido pelo roscore. Cada nó tem a opção de se inscrever

em tópicos ou publicar tópicos sobre trocas de informações. Um conjunto de nós geralmente é agrupado em um pacote que contém bibliotecas, executáveis e código-fonte. A arquitetura ROS foi projetada e dividida em três seções ou níveis conceituais, Sistemas de Arquivos, Processamento Grafo e Comunidade.

2.4.1 Sistemas de Arquivos (Filesystem level)

O primeiro nível é o nível do sistema de arquivos. Neste nível, um grupo de conceitos é usado para explicar como o ROS é formado internamente, a estrutura da pasta e os arquivos mínimos que ele precisa para funcionar. Semelhante a um sistema operacional, os arquivos ROS também são organizados no disco rígido de uma forma específica. Neste nível, podemos ver como esses arquivos estão organizados no disco. A seguir, os principais termos utilizados:

- Pacotes: São a unidade individual da estrutura do software ROS. Um pacote ROS pode conter código fonte, bibliotecas de terceiros, configuração de arquivos e assim por diante. Pacotes ROS podem ser reutilizados e compartilhados.

- Manifestos do pacote: O arquivo manifests (package.xml) terá todos os detalhes dos pacotes, incluindo o nome, descrição, licença e, mais importante, as dependências do pacote.

- Tipos de mensagem (msg): As descrições de mensagens são armazenadas na pasta msg em um pacote. As mensagens ROS são estruturas de dados para o envio de dados através do sistema de passagem de mensagens do ROS. As definições de mensagem são armazenadas em um arquivo com a extensão .msg.

- Tipos de serviço (srv): As descrições de serviço são armazenadas na pasta srv com a extensão .srv. O arquivo srv define a estrutura de dados de solicitação e resposta para o serviço no ROS.

2.4.2 Processamento de grafo

O Processamento de Grafo do ROS é a rede peer-to-peer do sistema ROS que processa dados. Os recursos básicos do Processamento de Grafo do ROS são nós, ROS Mestre, servidor de parâmetros, tópicos, mensagens, serviçose bolsas:

- Nós (nodes): o nó ROS é um processo que usa funcionalidades ROS para processar os dados. Um nó basicamente calcula. Por exemplo, um nó pode processar os dados de um sensor laser para verificar se existe alguma colisão. Um nó ROS é escrito com a ajuda de uma biblioteca cliente ROS (como roscpp e rospy).

- ROS Mestre (master): Os nós ROS podem se conectar uns aos outros usando um programa chamado ROS Mestre. Isso fornece o nome, o registro e a consulta ao restante do gráfico de computação. Sem iniciar o mestre, os nós não se encontrarão e não poderão enviar mensagens.

- Servidor de parâmetros (parameter server): Os parâmetros do ROS são valores estáticos que são armazenados em um local global chamado servidor de parâmetros onde todos os nós podem acessar esses valores. Podemos até definir o escopo do servidor de parâmetros como privado ou público para que ele possa acessar um nó ou acessar todos os nós.

- Tópicos (topics) ROS: Os nós ROS se comunicam entre si usando um barramento chamado tópico ROS. Os dados fluem através do tópico na forma de mensagens. O envio de mensagens sobre um tópico é chamado de publicação e o recebimento dos dados por meio de um tópico é chamado de subscrição.

- Mensagens(messages): Uma mensagem ROS é um tipo de dados que consiste em tipos de dados primitivos, como inteiros, pontos flutuantes e Booleanos. As mensagens do ROS fluem pelo tópico do ROS. Um tópico só pode enviar / receber um tipo de mensagem de cada vez. Podemos criar nossa própria definição de mensagem e enviá-la através dos tópicos.

- Serviços (services): O modelo de publicação/assinatura usando tópicos do ROS é uma maneira muito fácil de se comunicar. Esse método de comunicação é um

modo de comunicação um-para-muitos, o que significa que um tópico pode ser subscrito por qualquer número de nós. Em alguns casos, também podemos exigir um tipo de interação de solicitação/resposta, que geralmente é usada em sistemas distribuídos. Esse tipo de interação pode ser feito usando serviços ROS. Os serviços ROS funcionam de maneira semelhante aos tópicos do ROS, pois possuem uma definição de tipo de mensagem. Usando essa definição de mensagem, podemos enviar a solicitação de serviço para outro nó que fornece o serviço. O resultado do serviço será enviado como resposta. O nó tem que esperar até que o resultado seja recebido do outro nó.

- Bolsas (bags): Estes são os formatos em que para guardar e reproduzir os temas ROS. As bolsas ROS são uma ferramenta importante para registrar os dados do sensor e os dados processados. Essas bolsas podem ser usadas posteriormente para testar nosso algoritmo offline.

O diagrama da Figura 2.2 mostra como os tópicos e serviços funcionam entre os nós e o mestre (LENTIN, 2018):

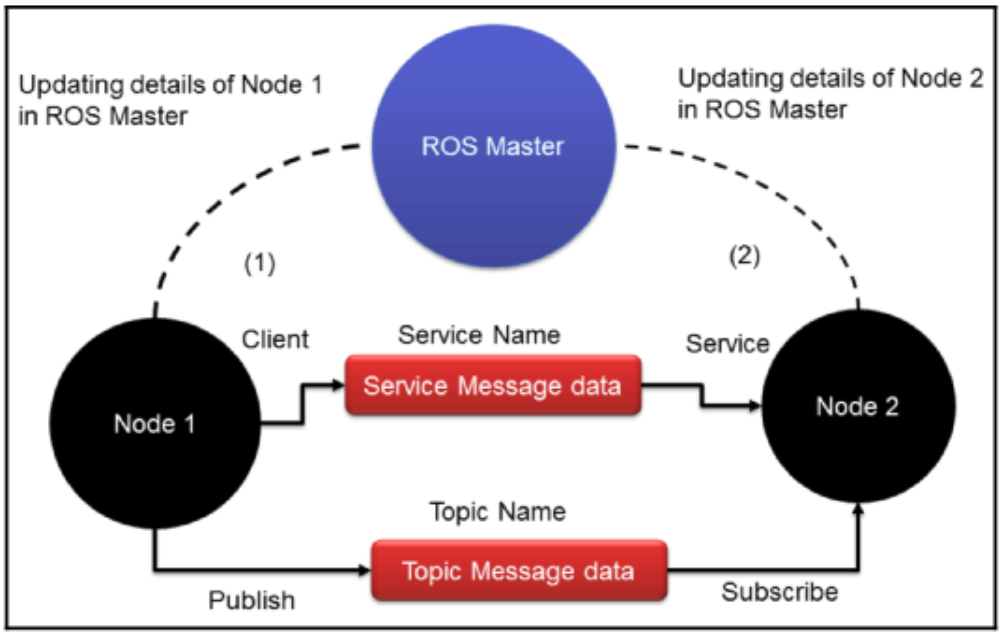


Figura 2.2: Comunicação entre os nós e o mestre Fonte: LENTIN2018

Para permitir a comunicação entre os nós 1 e 2 precisa iniciar o ROS Mestre.

No exemplo o nó 1 informa o Mestre que irá publicar um tópico com um determinado tipo de mensagem. Quando o nó 2 deseja subscrever o tópico o Mestre compartilhará as informações recebidas do nó 1 e alocará uma porta para iniciar a comunicação diretamente entre os dois nós. Os serviços ROS funcionam da mesma maneira.

2.4.3 Comunidade

A comunidade ROS consiste em desenvolvedores e pesquisadores de ROS que podem criar e manter pacotes e trocar novas informações relacionadas a pacotes existentes, pacotes recém-lançados e outras notícias relacionadas à estrutura ROS. A comunidade ROS fornece os seguintes serviços:

- Distribuições: Uma distribuição ROS tem um conjunto de pacotes que vêm com uma específica versão. Existem diversas versões disponíveis, como o ROS Indigo, Lunar, Kinetic e Melodic que podem ser instaladas. Na maioria dos casos, os pacotes dentro de uma distribuição são relativamente estáveis.

- Repositórios: Os repositórios online são os locais onde podem ser encontrados os pacotes. Normalmente, os desenvolvedores mantêm um conjunto de pacotes semelhantes chamados meta packages em um repositório. Também pode se manter um pacote individual em um único repositório.

- O wiki do ROS: O wiki do ROS é o local onde a documentação do ROS está disponível. É possível aprender sobre o ROS, desde seus conceitos mais básicos até a programação mais avançada, usando o wiki ROS (http: // wiki.ros.org).

- crição na lista de discussão do ROS (http://lists.ros.org/mailman/listinfo/ros-users). As últimas notícias do ROS podem ser obtidas em https://discourse.ros.org.

- Respostas do ROS: podem ser feitas perguntas relacionadas ao ROS no portal https://answers.ros.org/questions/ e obter suporte de desenvolvedores de todo o mundo.

Para mais informações, é possível consultar o site oficial do Ros em www.ros.org

2.4.4 Instalação e criação do ambiente de trabalho

A instalação do ROS permite criar o ambiente onde serão desenvolvidas as tarefas de criação e manipulação de diferentes robos. Nesse trabalho o ROS, na versão Indigo foi instalado em uma distribuição Ubuntu 14.04. Os detalhes da instalação e da criação do ambiente de trabalho podem ser encontrados na APÊNDICE I

2.4.5 Nodos do ROS

Um nó é um arquivo executável dentro de um pacote ROS. Os nós ROS usam uma biblioteca de clientes ROS para se comunicar com outros nós. Os nós podem publicar ou assinar um tópico. Os nós também podem fornecer ou usar um Serviço. As bibliotecas de clientes ROS permitem que os nodos escritos em diferentes linguagens de programação se comuniquem, as mais usadas rospy, biblioteca do cliente python e roscpp, biblioteca de clientes c ++.

2.4.6 Comandos do ROS

2.4.6.1 Roscore

Roscore é um conjunto de nodos e programas que são pré-requisitos para um sistema ROS, deve ser executado para que os nodos do ROS possam se comunicar. É lançado com o comando roscore que inicia ROS master, ROS parameter server e rosout. ROS master é um nodo que provê serviços de registro e consulta de nomes de nodos, tópicos e serviços, o ROS parameter service provê um dicionário acessível aos demais nodos e rosout é o tópico padrão do ROS para publicação de mensagens de log.

2.4.6.2 Rosnode

Abrindo um novo terminal usando *rosnode* pode ser verificado o que o *roscore* executou com *rosnode list* ou obter informações sobre um nó específico com *rosnode info*.

2.4.6.3 Rosrun

rosrun permite usar o nome do pacote para executar diretamente um nó dentro de um pacote (sem ter que especificar o caminho do pacote).

2.4.6.4 Uso dos comandos ROS

Uma aplicação do uso dos comandos *roscore, rosnode* e *rosrun* encontra-se no APÊNDICE II

2.5 Gazebo/ROS

O Gazebo é um ambiente de simulação de robôs gratuito e de código aberto desenvolvido pela Willow Garage. Como uma ferramenta multifuncional para desenvolvedores de robôs ROS, o Gazebo suporta o seguinte:

- Design de modelos de robôs
- Prototipagem rápida e teste de algoritmos
- Teste de regressão usando cenários realistas
- Simulação de ambientes internos e externos
- Simulação de dados de sensores para telêmetros a laser, câmeras 2D / 3D, sensores estilo kinect, sensores de contato, torque de força e muito mais
- Objetos 3D avançados e ambientes utilizando Object-Oriented Graphics Rendering Engine (OGRE)
- Vários mecanismos de física de alto desempenho (Open Dynamics Engine (ODE), Bullet, Simbody e Kit de Ferramentas de Animação Dinâmica e Robótica (Dynamic Animation and Robotics Toolkit DART) para modelar a dinâmica do mundo real.

O Gazebo normalmente já vem instalado no ROS.

3 METODOLOGIA

Para um correto estudo das propriedades e das aplicações dos robôs manipuladores, torna-se necessário aprofundar o planejamento de caminho. Nesse capítulo serão apresentadas as ferramentas e bibliotecas que viabilizara a execução deste trabalho.

3.1 Equipamento utilizado

Para o desenvolvemento do trabalho utilizou-se um notebook da marca DEll , modelo Inspiron 5537, que possui 16 Gb de memória RAM, processador Intel Core i7-4500U CPU 1.80GHz, placa grafica Intel Graphics 4400.

3.2 Linguagem de programação utilizada

As principais bibliotecas clientes de ROS (C++, Python, Lisp) são voltadas para um sistema do tipo Unix, devido, principalmente, a sua dependência de grandes coleções de software de código aberto. Para o desenvolvimento do trabalho foi utilizada a linguagem de programação Python e a biblioteca Rospy disponibilizada pelo ROS.

3.2.1 Python

Python é, provavelmente, a linguagem de programação popular mais fácil e agradável de lidar. O código Python é simples para ler e escrever, e consegue ser conciso sem ser algo enigmático (SUMMERFIELD, 2013). É uma linguagem multiplataforma: em geral, o Python pode ser executado no Windows e em sistemas derivados do Unix, o presente trabalho foi desenvolvido em ambiente Ubuntu. Um dos pontos fortes do Python é que ele vem com uma biblioteca padrão bastante completa e utiliza a biblioteca cliente rospy.

3.2.1.1 Rospy

Rospy é uma biblioteca cliente Python pura para o ROS. A API(Application Programming Interface) do cliente rospy permite que os programadores do Python façam

uma interface rápida com os Tópicos, Serviços e Parâmetros do ROS. O design do rospy favorece a velocidade de implementação (ou seja, o tempo do desenvolvedor) sobre o desempenho do tempo de execução, para que os algoritmos possam ser rapidamente prototipados e testados no ROS. Também é ideal para código de caminho não crítico, como código de configuração e inicialização. Muitas das ferramentas do ROS são escritas em rospy para aproveitar os recursos de introspecção do tipo. Muitas das ferramentas ROS, como rostopic e rosservice, são construídas usando o rospy.

3.3 Ambiente de simulação - Gazebo

O Gazebo é um simulador de ambientes 3D que possibilita avaliar o comportamento de um robô em um mundo virtual. Ele permite, entre muitas outras opções, projetar robôs de forma personalizada, criar mundos virtuais usando ferramentas CAD simples e importar modelos já criados.

Além disso, é possível sincronizá-lo com o ROS para que os robôs emulados publiquem as informações de seus sensores nos nós, além de implementar uma lógica e um controle que dá ordens ao robô.

3.4 Planejador de caminhos - Moveit

O MoveIt! é um software utilizado para a manipulação móvel, incorporando os últimos avanços em planejamento de movimento, manipulação, percepção 3D, cinemática, controle e navegação. Ele fornece uma plataforma fácil de usar para o desenvolvimento de aplicativos avançados de robótica, avaliando novos projetos de robôs e construindo produtos robóticos integrados para domínios industriais, comerciais, de pesquisa e desenvolvimento e outros (MOVEIT, 2018). MoveIt! é o software de código aberto mais utilizado para manipulação e é empregado para o planejamento de caminhos em mais de 65 tipos de robôs diferentes.

3.4.1 Sistema Moveit. Relação do Moveit e Ros

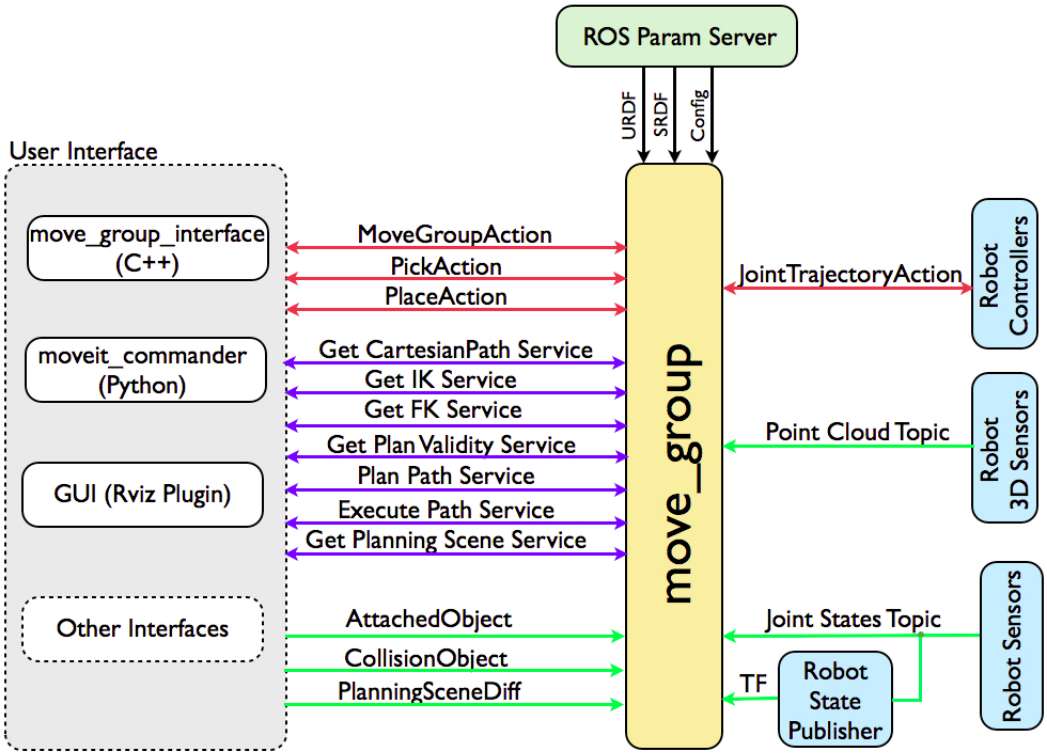


Figura 3.1: Arquitetura do MoveIt! Fonte: MOVEIT2018

O nó move_group

A Figura 3.1 mostra a arquitetura do sistema de alto nível para o nó primário fornecido pelo *MoveIt!* chamado *move_group*. Esse nó serve como um integrador: reunindo todos os componentes individuais para fornecer um conjunto de ações e serviços do ROS a serem aplicados pelos usuários.

Interface de usuário (User Interface)

Os usuários podem acessar as ações e serviços fornecidos por uma das três maneiras:

- Em C++ - usando o pacote *move_group_interface* que fornece uma interface C ++ fácil de configurar

- Em Python - usando o pacote *moveit_commander*
- Através de uma GUI - usando o plugin Motion Planning para o Rviz (o visualizador ROS)

o nó pode ser configurado usando um servidor de parâmetros (Param Server) do ROS de onde também obterá o URDF(Unified Robot Description Format) e o SRDF(Semantic Robot Description Format) para o robô.

Configuração (Configuration)

O *move_group* é um nó ROS que utiliza o Param Server, para obter três tipos de informações:

- URDF (Unified Robot Description Format), formato XML para representar um modelo de robô. Procura o parâmetro *robot_description* no ROS param server para obter o URDF do robô.
- SRDF (Semantic Robot Description Format), representação de informação semântica sobre robôs. Procura o parâmetro *robot_description_semantic* no ROS param server para obter o SRDF para o robô. O SRDF é normalmente criado (uma vez) por um usuário usando o MoveIt! Setup Assistant.
- MoveIt! configuração.O nó procurará no ROS param server por outra configuração específica para MoveIt! incluindo limites de articulação, cinemática, planejamento de movimento, percepção e outras informações. Arquivos de configuração para esses componentes são gerados automaticamente pelo MoveIt! Setup Assistant e armazenado no diretório de configuração do correspondente pacote de configuração MoveIt! para o robô.

Interface de Robô (Robot Interface)

O nó comunica com o robô através de tópicos. Por meio desse tipo de comunicação é possível obter informações sobre o estado atual do robô (posições das articulações, etc.), informações dos sensores, além de trocar informações com os seus controladores.

Informação do estado da Articulação (Joint State Information)

O *move_group* utiliza o tópico *joint_state*s para obter o estado atual (posições das articulações, etc.), conjuntos de pontos e outros dados dos sensores do robô, além de transmitir informações para os controladores do robô (por exemplo, editores separados podem ser usados para o braço e base móvel de um robô). Pode ser usado em conjunto com o nó *robot_state_publisher* para também publicar transformações para todos os estados de junção.

Converter Informação (Transform Information)

O *move_group* controla a conversão das informações usando a biblioteca ROS TF. Isso permite obter informações globais sobre a posição do robô (entre outras coisas). Também pode usar o TF para descobrir essa conversão para uso interno. Cabe ressaltar que o nó apenas sobrescreve o tópico TF, exigindo que o robô publique o seu estado.

Interface do Controlador (Controller Interface)

O *move_group* utiliza o tópico *joint_states* para obter o estado atual (posições das articulações, etc.), conjuntos de pontos e outros dados dos sensores do robô, além de transmitir informações para os controladores do robô.

Cena de planejamento (Planning Scene)

O *Planning Scene Monitor* é usado para manter uma cena de planejamento, que é uma representação do mundo e do estado atual do robô. O estado do robô pode incluir quaisquer objetos transportados pelo robô que sejam considerados rigidamente conectados ao robô. Mais detalhes sobre a arquitetura para manter e atualizar a cena do planejamento podem ser encontrados na Figura 3.3.

Recursos Extensíveis (Extensible Capabilities)

O sistema é estruturado para ser facilmente extensível, recursos individuais como transferências de objetos, cinemática, planejamento de movimento são realmente imple-

mentados como plugins separados com uma classe base comum. Os plugins são configuráveis usando o ROS através de um conjunto de parâmetros yaml (linguagem de marcação) do ROS e através do uso da biblioteca pluginlib do ROS. Não é necessário configurar plugins, já que eles são configurados automaticamente nos arquivos de inicialização gerados pelo *MoveIt! Setup Assistant*.

3.4.2 Planejamento de Movimento (Motion Planning)

MoveIt! trabalha com planejadores de movimento através de uma interface de plugins. Isso permite que o *MoveIt!* se comunique e utilize diferentes planejadores de movimento oferecidos por várias bibliotecas, tornando o *MoveIt!* facilmente extensível. A interface para os planejadores de movimento é por meio de uma ação ou serviço do ROS. Os planejadores de movimento padrão são configurados usando a biblioteca OMPL e o *MoveIt!* pelo MoveIt! Setup Assistant.

Solicitação do plano de movimento (Motion Plan Request)

A solicitação do plano de movimento especifica o que o planejador de movimento deve fazer. Normalmente, será pedido ao planejador de movimento para mover um braço para um local diferente (no espaço da articulação) ou determinar uma nova interação com o ambiente. As colisões são verificadas por padrão (incluindo colisões próprias). É possível anexar um objeto ao end efector (dispositivo no final de um braço robótico, projetado para interagir com o ambiente), por ex. se o robô pegar um objeto. Isso permite que o planejador de movimento contabilize o movimento do objeto durante o planejamento de caminhos. É possível especificar restrições para o planejador de movimento verificar. As restrições internas fornecidas pelo MoveIt! são restrições cinemáticas:

- Restrições de posição: restringem a posição de um link a uma região do espaço
- Restrições de orientação: restringem a orientação de um link dentro dos limites especificados de rolagem, inclinação ou mudanças de direção
- Restrições de visibilidade: restringem um ponto em um link para ficar dentro do cone de visibilidade de um sensor específico

- Restrições de articulações: restringem uma articulação entre dois valores
- Restrições especificadas pelo usuário: podem ser especificadas particulares restrições com um retorno de chamada definido pelo usuário.

Resultado do plano de movimento (The Motion Plan Result)

O nó *move_group* gera uma trajetória desejada em resposta à sua solicitação de plano de movimento. Esta trajetória moverá o braço (ou qualquer grupo de articulações) para o local desejado. O resultado é uma trajetória e não apenas um caminho, usando as velocidades máximas e as acelerações desejadas para gerar uma trajetória que obedeça à restrições de velocidade e aceleração no nível da articulação.

Pipeline de planejamento de movimento: planejadores de movimento e adaptadores de solicitação de plano (The Motion Planning Pipeline: Motion planners and Plan Request Adapters)

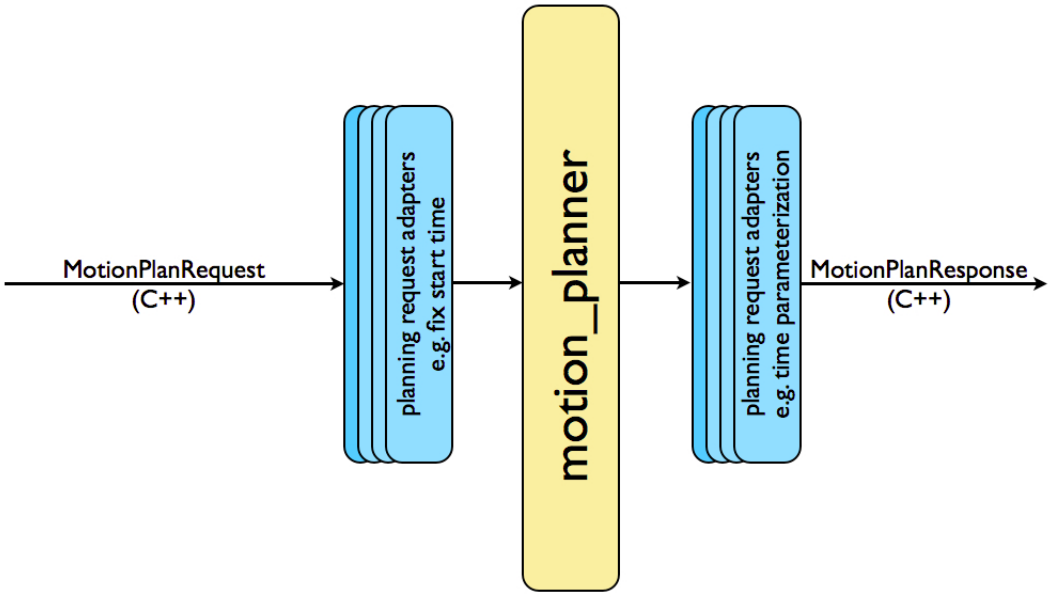


Figura 3.2: Pipeline de planejamento de movimento Fonte: MOVEIT2018

O pipeline de planejamento de movimento (Figura3.2) encadeia um planejador de movimento com outros componentes chamados de adaptadores de solicitação de planejamento. Os adaptadores de solicitação de planejamento permitem o pré-processamento das solicitações do plano de movimento e o pós-processamento das respostas do plano de movimento. O pré-processamento é útil em várias situações, por exemplo quando um estado de partida para o robô estiver ligeiramente fora dos limites de articulação especificados para o robô. O pós-processamento é necessário para várias outras operações, por exemplo para converter caminhos gerados por um robô em trajetórias parametrizadas pelo tempo. O *MoveIt!* fornece um conjunto de adaptadores de planejamento de movimento que executam individualmente suas funções específicas (OMPL).

OMPL

OMPL (Open Motion Planning Library) é uma biblioteca de planejamento de movimento de código aberto que implementa principalmente planejadores de movimentos aleatórios. *MoveIt!* integra-se diretamente com o OMPL e usa os planejadores de movimento dessa biblioteca como seu conjunto de planejadores padrão. Os planejadores em OMPL são abstratos, ou seja, OMPL não tem representação de robô. Em vez disso, *MoveIt!* configura o OMPL e fornece as informações de retorno.

Cena de planejamento (Planning Scene)

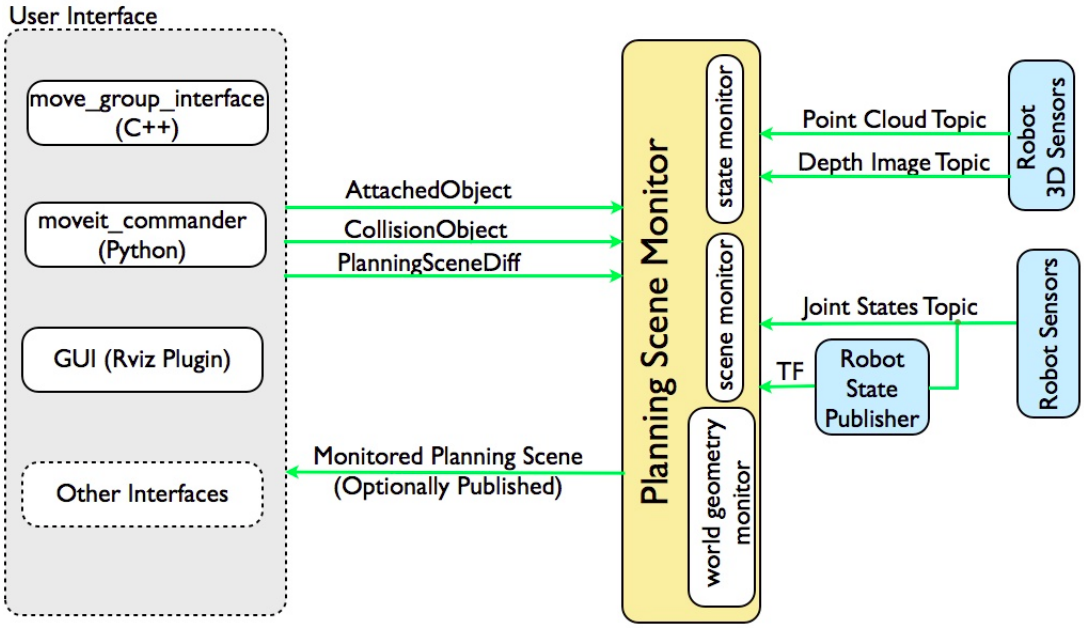


Figura 3.3: Cena do planejamento Fonte: MOVEIT2018

A cena de planejamento é usada para representar o mundo ao redor do robô e também armazena o estado do próprio robô. Ela é mantida pelo monitor de cena de planejamento dentro do nó do grupo de movimentação. O monitor de cena de planejamento escuta:

- Informações do Estado: no tópico *joint_state*
- Informações do sensor: usando o monitor de geometria do mundo descrito abaixo
- Informações de geometria do mundo: da entrada do usuário no tópico planning_scene (cena de planejamento).

Monitor de Geometria do mundo

O monitor de geometria do mundo constrói a geometria do mundo usando informações dos sensores no robô e das entradas do usuário. Ele usa o monitor do mapa de ocupação descrito abaixo para construir uma representação 3D do ambiente ao redor do robô e aumenta isso com informações no tópico *planning_scene* para adicionar informações sobre objetos.

Percepção 3D (3D Perception)

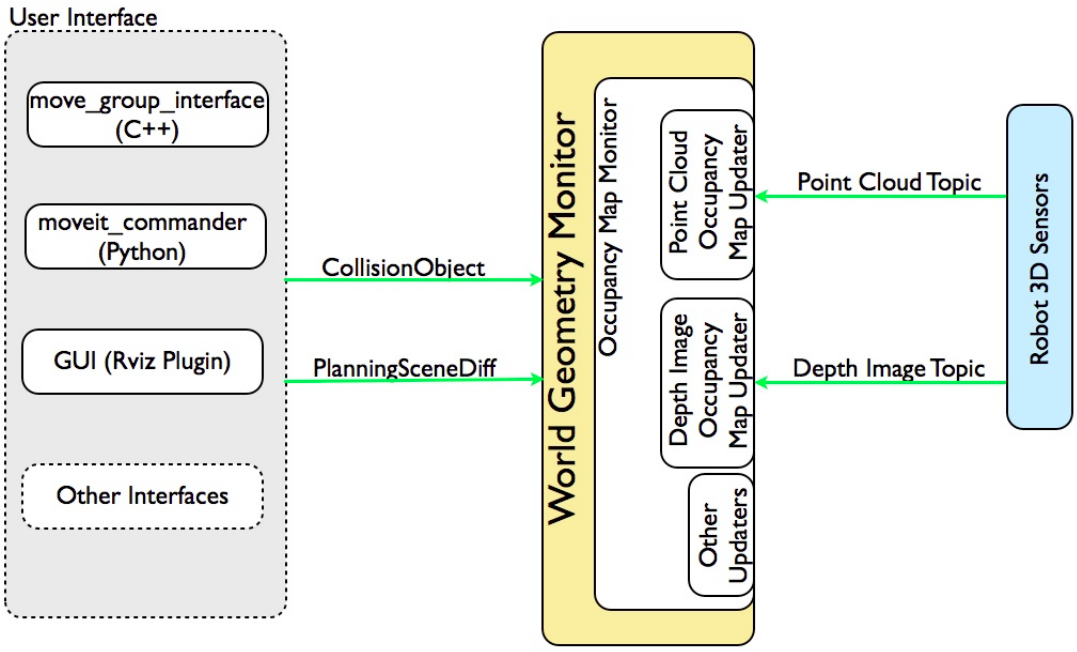


Figura 3.4: Percepção 3D em MoveIt! Fonte: MOVEIT2018

Percepção 3D em MoveIt! é manipulado pelo monitor do mapa de ocupação. O monitor do mapa de ocupação usa uma arquitetura de plugin para lidar com diferentes tipos de entrada do sensor, conforme mostrado na Figura 3.4. Em particular, MoveIt! tem suporte embutido para lidar com dois tipos de entradas:

- Nuvens de pontos (point cloud): manipuladas pelo plugin de atualização do mapa de ocupação de nuvem de pontos
- Imagens de profundidade: manipuladas pelo plugin de atualização do mapa de ocupação de imagem de profundidade

Podem ser adicionados tipos de atualizadores como um plugin ao monitor do mapa de ocupação (occupancy map monitor).

Octomap

O monitor de mapa de ocupação usa um Octomap para manter o mapa de ocupação do ambiente. O Octomap pode realmente codificar informações probabilísticas sobre

células individuais, embora essas informações não sejam usadas atualmente no MoveIt. O Octomap pode ser passado diretamente para o FCL (Flexible Collision Library), a biblioteca de checagem de colisões utilizadas no Moveit

Atualizador do Mapa de Ocupação da Imagem de Profundidade (Depth Image Occupancy Map Updater)

O atualizador de mapa de ocupação de imagem de profundidade inclui seu próprio filtro automático, ou seja, removerá partes visíveis do robô do mapa de profundidade. Ele usa informações atuais sobre o robô (o estado do robô) para realizar esta operação.

3.4.3 O plugin da cinemática

MoveIt! usa uma infraestrutura de plugins, especialmente voltada para permitir que os usuários escrevam seus próprios algoritmos de cinemática inversa. A cinemática avançada e a localização de jacobianos estão integradas na própria classe RobotState. O plugin de cinemática inversa padrão para MoveIt! é configurado usando o KDL (Kinematic and Dynamic Solvers) que utiliza o determinante da matriz jacobiana. Este plugin é configurado automaticamente pelo MoveIt! Setup Assistant.

3.4.3.1 Plugin IKFast

Frequentemente, os utilizadores podem optar por implementar as suas próprias soluções cinemáticas, por exemplo o robô PR2 tem seus próprios solucionadores de cinemática. Uma abordagem popular para implementar esse solver é usar o pacote IKFast para gerar o código C ++ ou Python, necessário para trabalhar com um robô específico.

3.4.4 Verificação da colisão (Collision Checking)

A verificação de colisão em MoveIt! é efetuada mediante uma cena de planejamento usando o objeto CollisionWorld. MoveIt! é configurado para que os usuários nunca precisem se preocupar sobre como a verificação de colisão está acontecendo. A verificação de colisão em MoveIt! é efetuada usando o pacote FCL (Flexible Collision

Library) do MoveIt!

MoveIt! suporta verificação de colisão para diferentes tipos de objetos, incluindo:

- Malhas
- Formas primitivas - por ex. caixas, cilindros, cones, esferas e planos
- Octomap - o objeto Octomap pode ser usado diretamente para verificação de colisão

Matriz de Colisão Permitida (Allowed Collision Matrix - ACM)

A checagem de colisão é uma operação computacional muito cara, geralmente responsável por quase 90% da despesa computacional durante o planejamento de movimento. A matriz de colisão permitida ou ACM codifica um valor binário correspondente à necessidade de verificar a colisão entre pares de corpos (que podem estar no robô ou no mundo). Se o valor correspondente a dois corpos for definido como 1 no ACM, isso especifica que uma verificação de colisão entre os dois corpos não é necessária. Isto aconteceria se, por exemplo, os dois corpos estão sempre tão longe que nunca colidiriam um com o outro.

3.4.5 Processamento de Trajetória

Os planejadores de movimento normalmente geram apenas "caminhos", ou seja, não há informações de tempo associadas aos caminhos. O Moveit!, entretanto, inclui uma rotina de processamento de trajetória que pode trabalhar nesses caminhos e gerar trajetórias que são apropriadamente parametrizadas no tempo para a máxima velocidade e limites de aceleração impostos em junções individuais. Esses limites são lidos de um arquivo especial joint_limits.yaml especificado para cada robô.

3.4.5.1 Assistente de configuração do MoveIt! (MoveIt! Setup Assistant)

O Assistente de Configuração do MoveIt! é uma interface gráfica do usuário para configurar qualquer robô para uso com o MoveIt!. Sua principal função é gerar um arquivo SRDF (Semantic Robot Description Format) característico do robô. Além disso, gera outros arquivos de configuração necessários para uso com o MoveIt! Pipeline.

O uso do Assistente de configuração é mostrado detalhadamente na APÊNDICE III.

3.4.6 BAXTER

O Baxter é um robô de dois braços criado pela Rethink Robotics em 2012, projetado para executar tarefas industriais simples, tais como carga, descarga e manuseio de materiais. Cada um dos braços do Baxter tem sete graus de liberdade (DOF) e uma série de atuadores comuns. O robô, mostrado na Figura 3.5, possui vários sensores que permitem realizar muitas tarefas:

- Um sensor de sonar de 360 graus no topo da cabeça da Baxter
- Uma face de tela de 1024 x 600 pixels com uma câmera embutida
- Uma câmera, sensor infravermelho e acelerômetro no punho no final de cada braço

Figura 3.5: O Robô Baxter Fonte: BAXTER2018

Uma segunda versão da Baxter foi introduzida pela empresa em 2013, ano seguinte ao lançamento da versão industrial. Esta segunda versão é destinada para o uso de organizações acadêmicas e de pesquisa. O hardware para a versão de pesquisa da Baxter é idêntico à versão de fabricação, no entanto, o software para as duas versões não é o mesmo. O robô de pesquisa do Baxter é configurado com um software de desenvolvimento (SDK) que é executado em uma estação de trabalho remota e permite que os pesquisadores desenvolvam software personalizado para a Baxter. O software fornece uma API (interface de programação de aplicativos) do ROS de código aberto para executar diretamente comandos e scripts do ROS. Qualquer estação de trabalho remota (na rede da Baxter) lança os nós ROS para conectar-se ao Baxter e controlar suas juntas e sensores. No presente trabalho foi utilizado o software específico de simulação fornecido gratuitamente pela empresa Rethink Robotics

3.4.7 Manipulação do Baxter

Para manipular o Baxter, na APÊNDICE IV, será explicado como criar uma área de trabalho de desenvolvimento do Baxter para seguir com um simples exemplo de manipulação.

3.4.8 Baxter, MoveIt! e Rviz

Um tutorial, na APÊNDICE V, descreve como trabalhar com o robô de pesquisa Baxter mediante o MoveIt!, a estrutura de planejamento de movimento do ROS.

4 - RESULTADOS.

O trabalho foi desenvolvido nas seguintes etapas:

- Estudo da Robótica;
- Aprendizado do ROS, Gazebo e linguagem de programação Python;
- Pesquisa sobre artigos, livros e trabalhos relacionados ao projeto;
- Aprofundamento específico sobre o Baxter Robô;
- Desenvolvimento de uma aplicação em Python relativo à manipulação de objetos;
- Execução de testes da aplicação em ambiente Gazebo;
- Edição final da aplicação

4.1 Desenvolvimento

Escopo da aplicação é executar uma simulação de manipulação com o robô Baxter que consiste em deslocar um objeto de uma posição para uma outra.

4.1.1 SDK - Software development kit

Para se comunicar e comandar o Baxter, devemos estabelecer a conexão entre o PC de desenvolvimento e o robô. Tendo uma configuração de rede adequada, o shell RSDK refere-se à configuração do ambiente ROS que aponta o PC para o ROS Master, registrando o IP (Internet Protocol), permitindo que possa ser localizado por outros processos. O SDK fornece o script baxter.sh, que ajuda a obter uma configuração mais rápida e fácil do ambiente ROS permitindo a comunicação. No APÊNDICE IV pode ser encontrado todo o procedimento para a configuração do ambiente, incluindo as modificações do script baxter.sh.

4.1.2 Mundo do Baxter no Gazebo

A aplicação foi desenvolvida utilizando o simulador Gazebo. O mundo do Baxter é a base onde a aplicação manipula o robô permitindo a transferência de um objeto de uma posição para outra. A geração do mundo do Baxter pode ser vista na Apêndice V.

4.1.3 Manipulação do objeto

Uma vez aberto no simulador Gazebo o mundo de Baxter (APÊNDICE V Figura V.2) com o arquivo .launch:

roslaunch baxter_gazebo baxter_world.launch

executamos o script python:

python ik_pick_and_place_final.py

As Figuras 4.1 até 4.8 mostram as etapas da execução, o códico fonte é mostrado na APÊNDICE VI.

4.1.3.1 Estado inicial

Chamando a função *load_gazebo_models()* será criada, na frente do robô, uma mesa sobre a qual serão geradas as três posições, representadas com diferentes cores. O objeto a ser manipulado, um cubo, encontra-se na posição inicial (Figura 4.1).

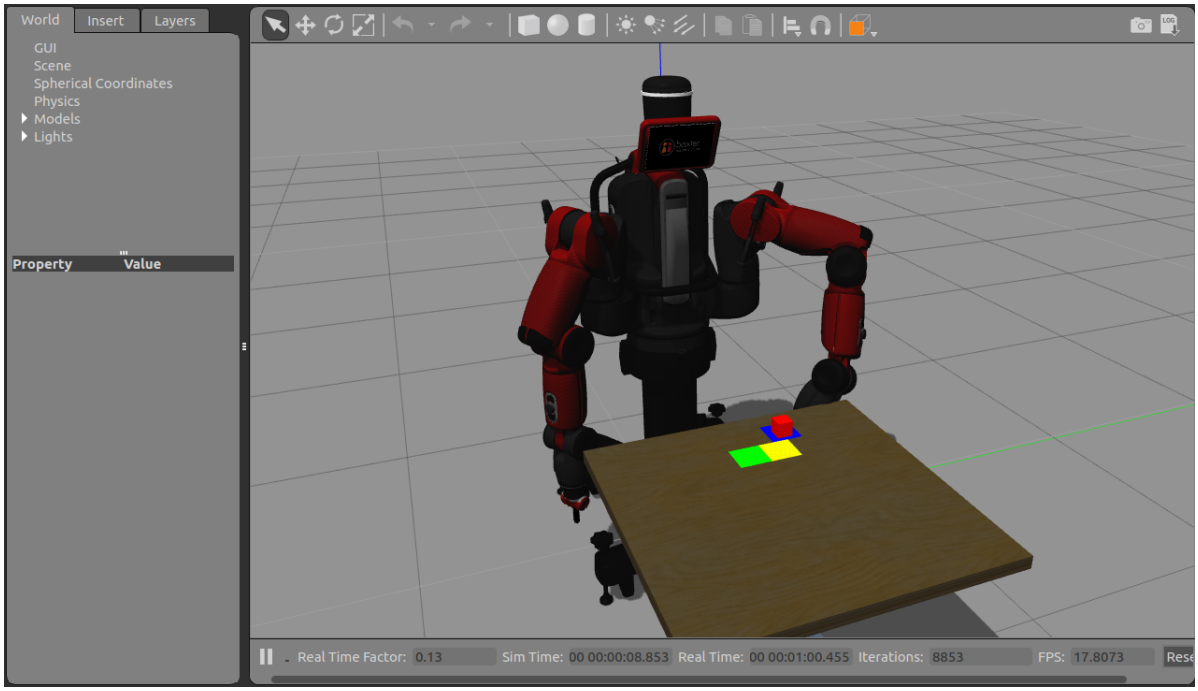


Figura 4.1: Estado inicial Fonte: Elaborado pelo autor

4.1.3.2 Garra posicionada acima do objeto

A função *move_to_start(starting_joint_angles)*, posiciona o braço esquerdo do robô perpendicular ao objeto com as garras abertas (Figura 4.2).

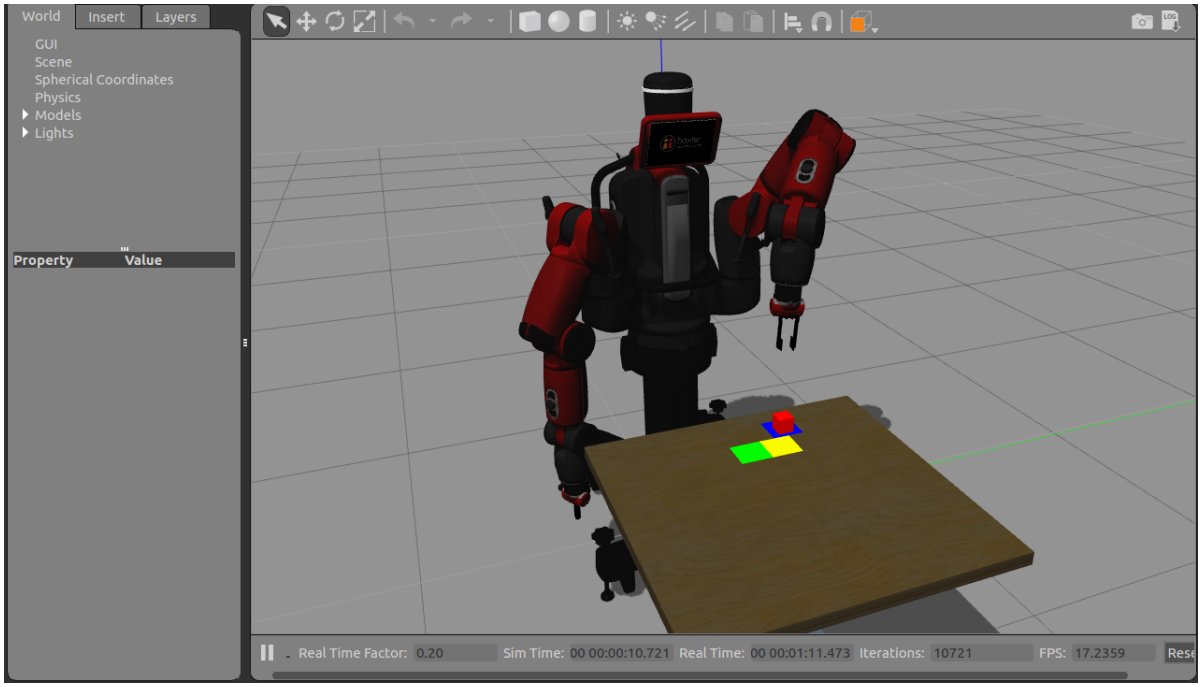


Figura 4.2: Garra posicionada acima do objeto Fonte: Elaborado pelo autor

4.1.3.3 Retirando objeto

A função *pick(block_poses[idx])*, faz o braço descer até o objeto, fecha as garras e inicia a transferência do objeto (Figura 4.3).

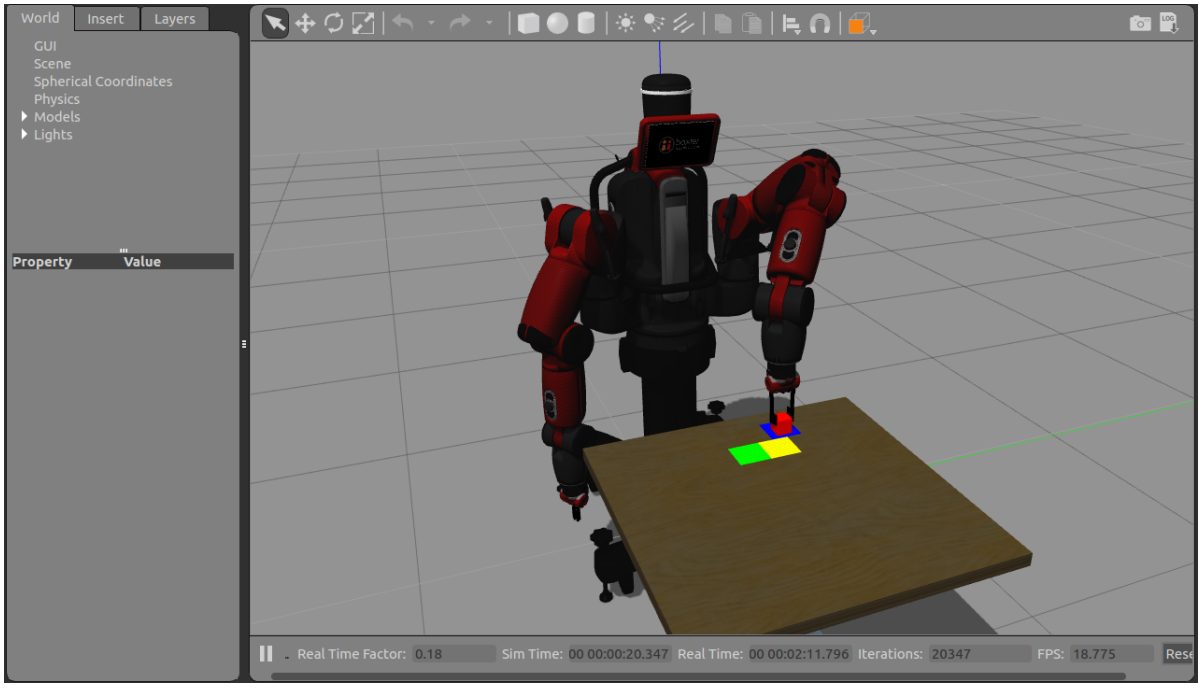


Figura 4.3: Retirando objeto Fonte: Elaborado pelo autor

4.1.3.4 Colocando o objeto no primeiro destino

Mediante a função *place(block_poses[idx])*, o braço se movimenta até se colocar logo acima da primeira posição, desce e deposita o objeto e levanta o braço novamente (Figura 4.4).

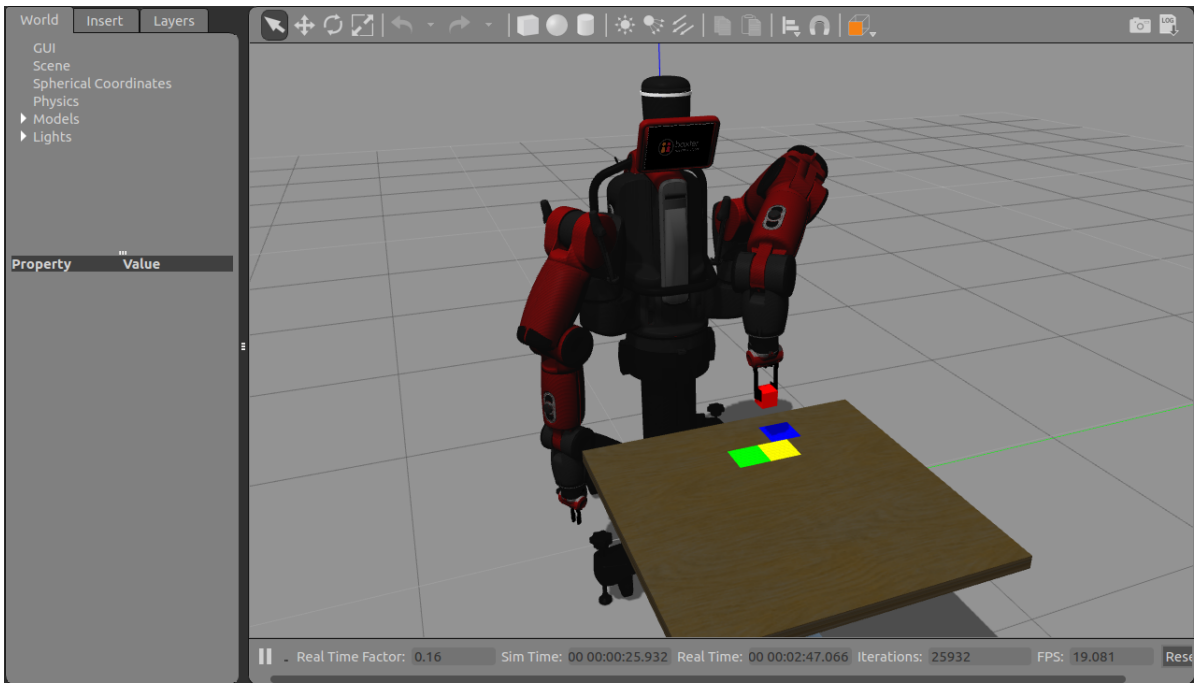


Figura 4.4: Colocando o objeto no primeiro destino Fonte: Elaborado pelo autor

4.1.3.5 Retirando objeto do primeiro destino

O braço desce novamente retirando o objeto (função *pick(block_poses[idx])*). A marca da primeira posição desaparece para mostrar que a etapa terminou (Figura 4.5).

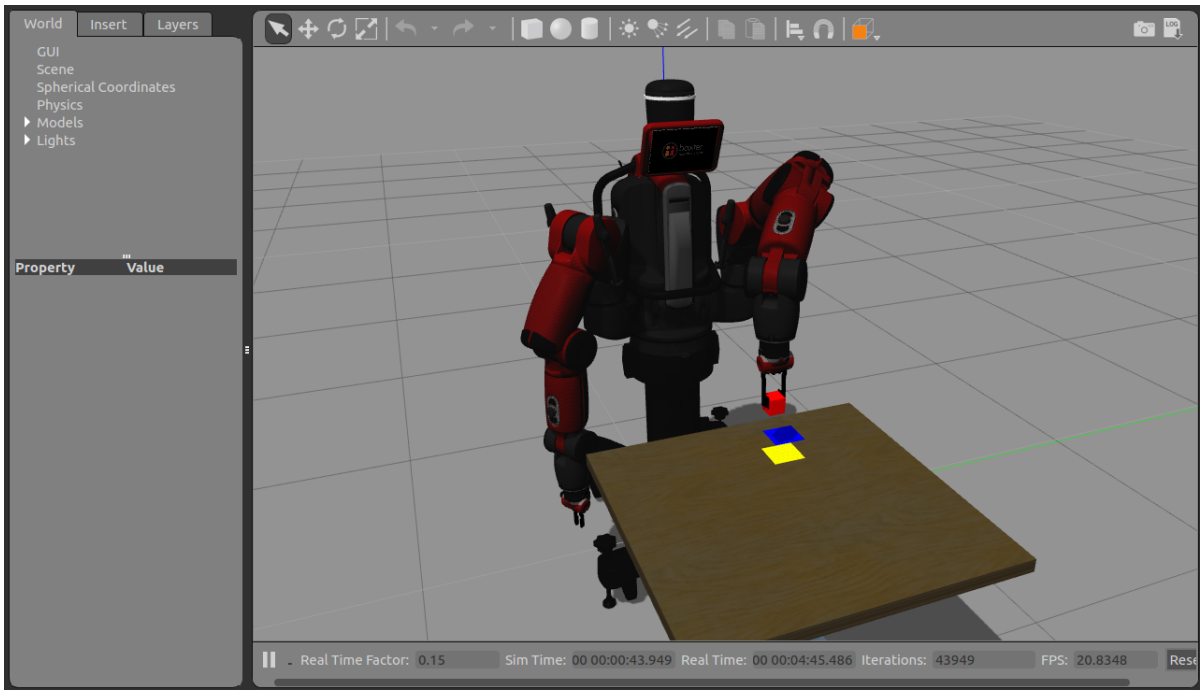


Figura 4.5: Retirando objeto do primeiro destino Fonte: Elaborado pelo autor

4.1.3.6 Colocando objeto no segundo destino

O braço deposita o objeto na segunda posição (função *pick(block_poses[idx])*). A marca da segunda posição desaparece para mostrar que a etapa terminou (Figura 4.6).

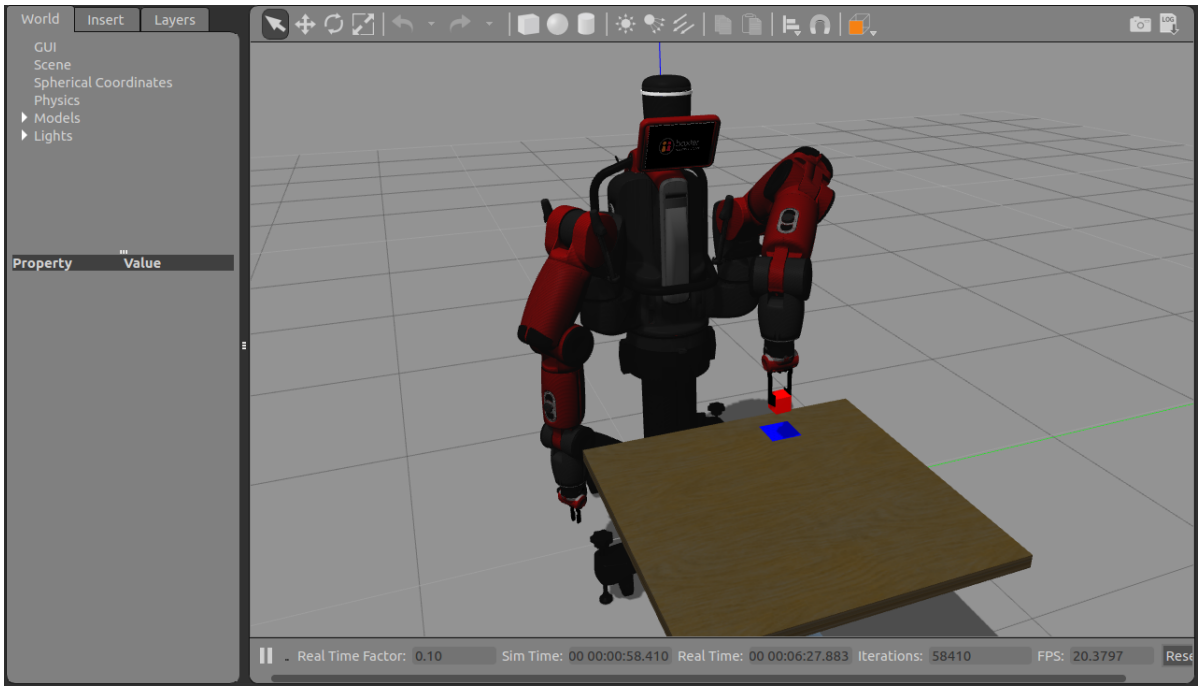


Figura 4.6: Retirando objeto do segundo destino Fonte: Elaborado pelo autor

4.1.3.7 Colocando objeto na posição final

O objeto é transferido para a posição final (Figura 4.7), coincidente com a posição de origem (função *place(block_poses[idx])*)

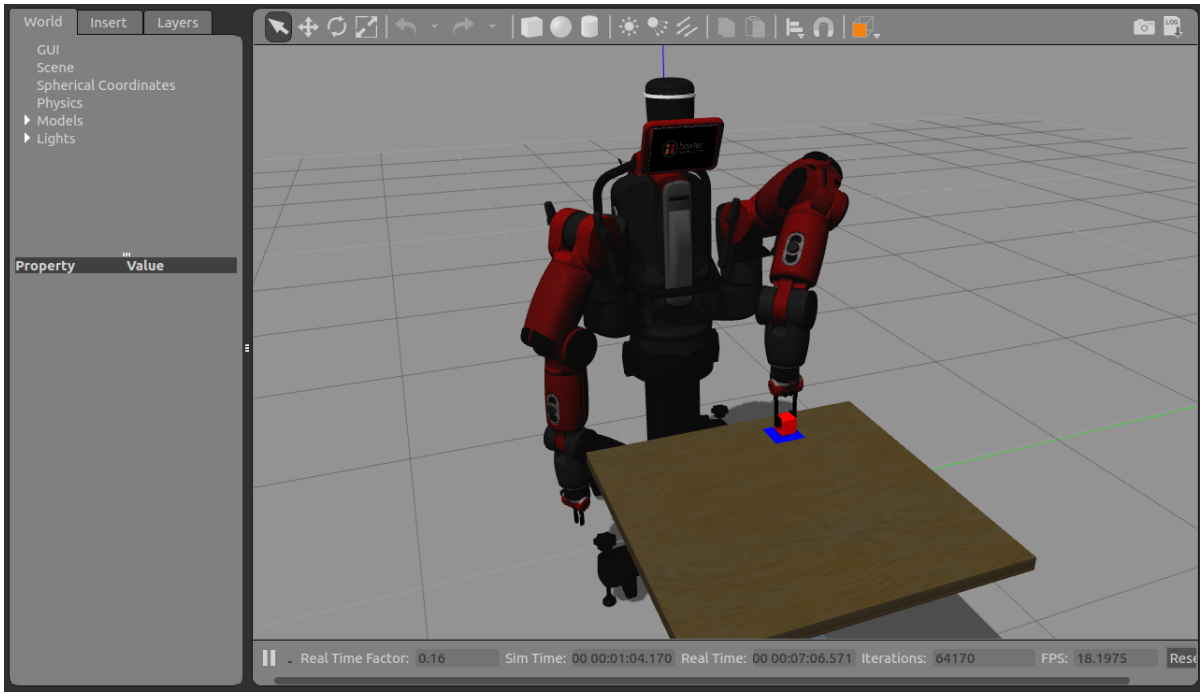


Figura 4.7: Colocando objeto na posição final Fonte: Elaborado pelo autor

4.1.3.8 Retornando ao estado inicial

Assim que o braço sobe a situação inicial é restabelecida (Figura 4.8) mediante a função *load_gazebo_models()*.

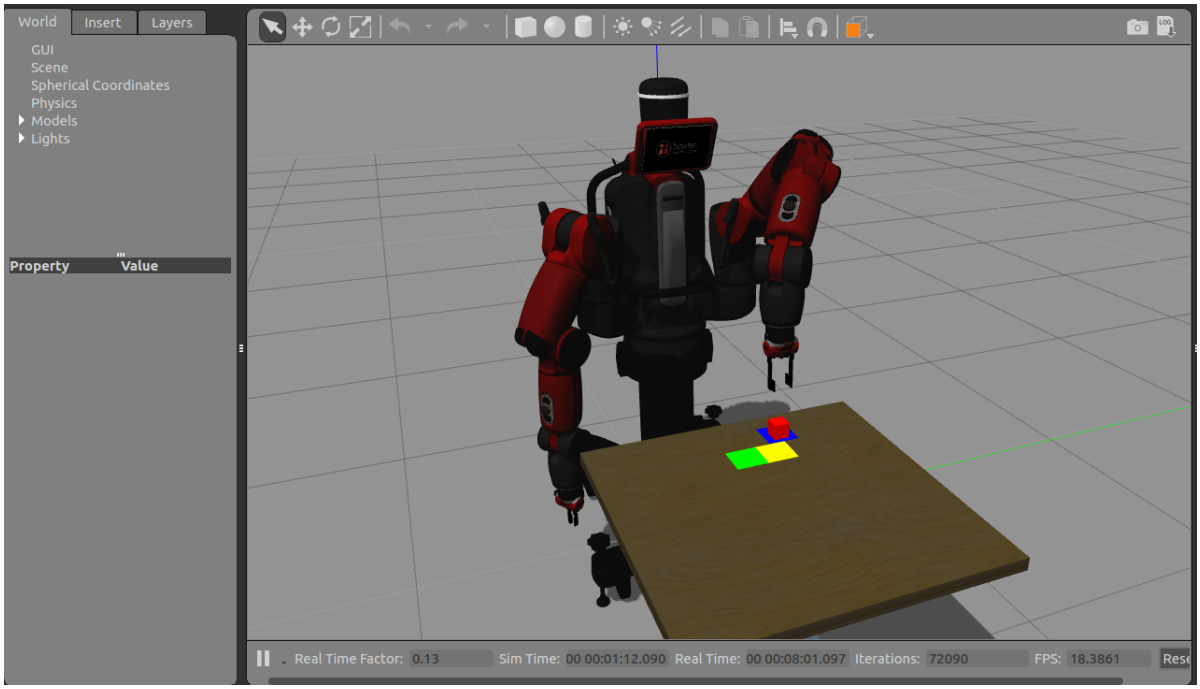


Figura 4.8: Retornando ao estado inicial Fonte: Elaborado pelo autor

5 - CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foram abordados temas que tratam do framework ROS, dos simuladores Gazebo e MoveIt! e do robô Baxter. Como linguagem de programação foi utilizado o Python. O estudo do ROS inicialmente apresenta algumas dificuldades mas, após pesquisas realizadas em livros específicos, obteve-se um melhor entendimento do sistema. Tendo, também, como base o material fornecido pela Rethink Robotics, construtora do robô, foi possível desenvolver uma aplicação para o controle do robô simulado utilizando o Gazebo e o MoveIT!. O trabalho desenvolvido constitui uma base de partida para futuras atividades de pesquisas sobre as múltiplas utilizações dos recursos fornecidos. Como primeira sugestão, podemos considerar um estudo sobre as possibilidades de uso, por parte dos robôs, de sensores que lidam com a observação de aspectos do mundo exterior ao robô como: sensores de contato, proximidade, visão, sensores laser, de ultrassom, de infravermelhos e químicos. Um outro interessante campo de estudo é relativo ao uso da inteligência artificial na robótica, suportadas pelas subdisciplinas associadas a estas áreas (Redes Neurais, Impressão 3D, Aprendizado de Máquina, Computação nas Nuvens, Realidade Aumentada, entre outras).

REFERÊNCIAS

BECKER, M. **Cinemática Direta de Manipuladores Robóticos**. 2008. Acesso 14 jul. 2016. Disponível em: <http://www.mecatronica.eesc.usp.br/wiki/upload/0/07/Aula5-SEM0317.pdf>.

CABRAL, E. **Cinemática da Posição de Robôs Manipuladores**. 2012. Acesso 18 ago. 2018. Disponível em: <http://sites.poli.usp.br/p/eduardo.cabral/CinemÃąticaDireta.pdf>.

FAIRCHILD, C.; HARMAN, T. L. **ROS Robotics By Example**. [S.l.: s.n.], 2016. 428 p. ISBN 9781782175193.

GAZEBO. **Gazebo Simulator**. 2017. Acesso 11 set. 2017. Disponível em: <http:gazebosim.org/>.

HONDA. **The Honda humanoid robot Asimo**. 2011. Acesso 22 mar. 2016. Disponível em: <http://asimo.honda.com/>.

LENTIN, J. **Learning Robotics Using Python**. [S.l.: s.n.], 2018. ISSN 1098-6596. ISBN 9788578110796.

LOPES. **Manipuladores de Estrutura Paralela**. 2002. Acesso 30 ago. 2017. Disponível em: <http://paginas.fe.up.pt/~aml/maic-files/Rob-para.pdf>.

MOVEIT. **Moveit concept**. 2018. Acesso 15 set. 2017. Disponível em: <https://moveit.ros.org/documentation/concepts/>.

NASA. **ROBONAUT 2**. 2014. Acesso 07 jan. 2017. Disponível em: <https://robonaut.jsc.nasa.gov/R2/>.

ROS. **Installation**. 2013. Acesso 12 jul 2017. Disponível em: <http://wiki.ros.org/ROS/Installation>.

ROS. **Introduction**. 2014. Acesso 18 nov 2016. Disponível em: <http://wiki.ros.org/ROS/Introduction>.

ROS. **About ROS**. 2016. Acesso em 2 jun. 2017. Disponível em: <http://www.ros.org/about-ros/>.

ROVER. **ROVER Mars Exploration**. 2018. Acesso 18 ago 2018. Disponível em: <https://www.nasa.gov/mission-pages/mer/index.html>.

SANTANA, A. M. **Localização e Planejamento de Caminhos para um Robô Humanóide e um Robô Escravo com Rodas .** 2007. 1– 76 p. Acesso 13 nov 2016. Disponível em: <https://repositorio.ufrn.br/jspui/bitstream/123456789/15169/1/AndreMS.pdf>.

SILVEIRA. **Um novo método de planejamento de caminho para robôs baseado em espuma probabilística**. 2017. Acesso 22 jun. 2017. Disponível em: <https://repositorio.ufrn.br/jspui/handle/123456789/24115>.

SPONG, M. W. **Robot Modeling and Control (Free ver.)**. [S.l.: s.n.], 2006. ISSN 0272-1708. ISBN 0471649902.

SUMMERFIELD, M. **Programação em Python 3**. [S.l.: s.n.], 2013. ISBN ISBN-978-85-7608-384-9.

TRONCO. **Robôs Industriais**. 2004. Acesso 02 mar. 2017. Disponível em: <https://edisciplinas.usp.br/mod/resource/view.php?id=1061035>.

USP. **Laboratório de dinâmica e simulação veicular**. 2014. Acesso em 25 feb. 2017. Disponível em: <http://www.usp.br/ldsv/>.

APÊNDICES

APÊNDICE I – INSTALAÇÃO DO ROS E CRIAÇÃO DO AMBIENTE DE TRABALHO

Procedimento utilizado para instalação da distribuição do ROS Indigo no Ubuntu 14.04

Organização do ambiente:

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

Configuração da chave para o servidor:

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

Atualização do Debian

sudo apt-get update

Instalação do ROS Indigo:

sudo apt-get install ros-indigo-desktop-full

Inicialização do Rosdep.

O Rosdep permite instalar facilmente as dependências do sistema para a fonte que se deseja compilar e é necessário para executar alguns componentes principais no ROS:

sudo rosdep init

rosdep update

Adicionar as variáveis de ambiente:

echo "source /opt/ros/indigo/setup.bash"

/.bashrc

source /.bashrc

O rosinstall é uma ferramenta de linha de comando, distribuída separadamente, usada com frequência no ROS. Permite baixar facilmente muitas árvores de origem para pacotes ROS com um comando:

sudo apt-get install python-rosinstall

Os pacotes instalados podem ser encontrados aqui:

http://repositories.ros.org/status_page/ros_indigo_default.html

Criação do ambiente de trabalho

Uma vez instalado o ROS, a etapa seguinte será a criação do ambiente de trabalho:

mkdir -p /catkin_ws/src

cd /catkin_ws/

catkin_make

O comando catkin_make é uma ferramenta para trabalhar com espaços de trabalho do catkin. Ao executá-lo pela primeira vez no espaço de trabalho, ele criará na subpasta 'src' um link CMakeLists.txt. No diretório catkin_ws encontram-se também as pastas 'build' e 'devel'. Dentro da pasta 'devel', existem agora vários arquivos setup.*sh. Em seguida será necessário criar os novos arquivos setup.*sh:

\source devel/setup.bash

Para garantir que espaço de trabalho seja adequadamente sobreposto pelo script de instalação, é preciso verificar se a variável de ambiente ROS_PACKAGE_PATH inclui o diretório atual:

echo $ROS_PACKAGE_PATH

A resposta deve ser algo parecido com:

/opt/ros/indigo/share:/opt/ros/indigo/stacks

Sistema de Arquivos ROS

Instalação do ros-tutorial

sudo apt-get install ros-indigo-ros-tutorials

Conceitos básicos do sistema de Arquivos ROS

- Pacotes: os pacotes são a unidade de organização de software do código ROS. Cada pacote pode conter bibliotecas, executáveis, scripts ou outras ferramentas.
- Manifestos (package.xml): Um manifesto é uma descrição de um pacote. Define dependências entre pacotes e captura meta informações sobre o pacote como versão, mantenedor, licença, etc.

Criando um Pacote Catkin

Para criar um pacote é necessário é necessário mudar para o diretório src do espaço de trabalho Catkin:

cd /catkin_ws/src

Usando o script catkin_create_pkg, será criada uma pasta chamada ?beginner_tutorials?com dependencias std_msgs, roscpp e rospy:

catkin_create_pkg beginner_tutorials std_msgs rospy roscpp

A pasta beginner_tutorials contém um package.xml e um CMakeLists.txt, parcialmente preenchido com as informações fornecidas.

Os comandos roscd beginner_tutorial e cat package.xml permitem acesso ao pacote. Conjunto mínimo de tags que precisam ser presentes no pacote:

- name - O nome do pacote
- versão - O número da versão do pacote (necessariamente inteiros separados por 3 pontos)

- description - Uma descrição do conteúdo do pacote
- maintainer - O nome da pessoa que mantém o pacote
- license - A licença de software (por exemplo, GPL, BSD, ASL) sob a qual o código é liberado.

Um pacote fictício deve conter:

name: pack.exemplo/name

version: 1.2.4/version

description: "Este pacote tem como finalidade"

maintainer email: fulano@email.com

maintainer: John Smith

licence: BSD

APÊNDICE II – COMANDOS ROSCORE, ROSNODE E ROSRUN

As informações a seguir foram extraídas de (ROS, 2016)

Abrir um terminal Ubuntu e inicializar o ROS com o seguinte comando:

roscore

Em outra janela de terminal, inicializar o nó do turtlesim:

rosrun turtlesim turtlesim_node

Aparecerá uma tartaruga

Em uma nova janela de terminal inicializar um nó em que podemos utilizar as setas do teclado para controlar a tartaruga:

rosrun turtlesim turtle_teleop_key

A seguinte mensagem aparecerá:

Use arrow keys to move the turtle.

Isso significa que é possível usar as setas para teleoperação da tartaruga. É necessário colocar as janela do turtlesim e do nó de teleoperação lado a lado. A janela do nó de teleoperação deve ser selecionada para que o nó capture o pressionar das teclas. Usando as setas do teclado será possível ver a tartaruga se mover. Na Figura II.1 é mostrado o resultado de tais ações.

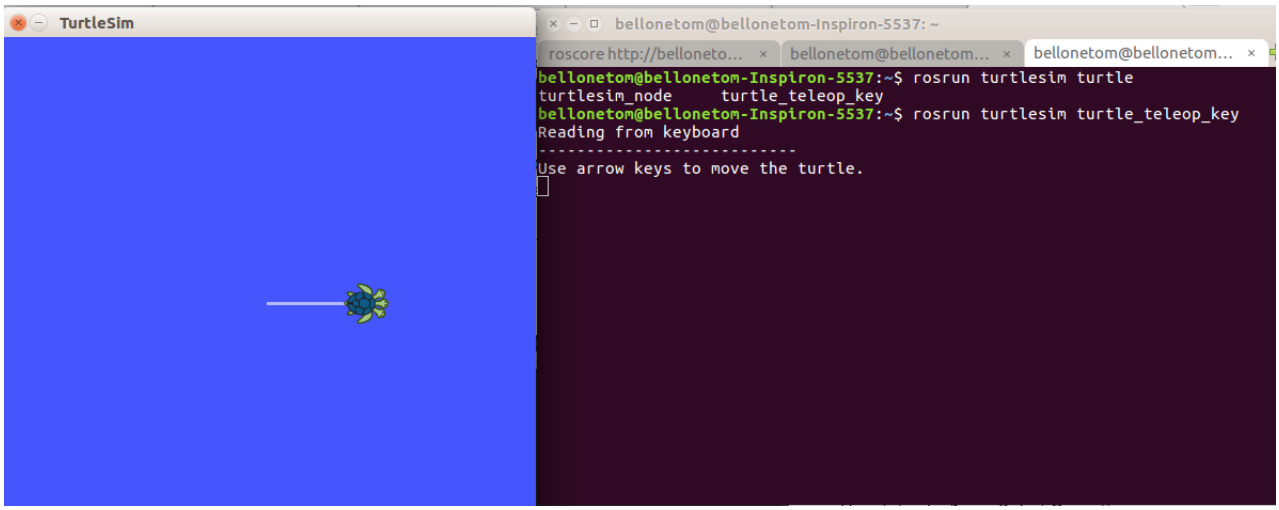


Figura II.1: Teleoperação no TurtleSim Fonte: ROS2014

Comando rostopic.

Esse comando permite realizar uma série de manipulações nos tópico do ROS. Serão utilizados cinco comandos:

- rostopic list: imprime uma lista com todos os tópicos que estão sendo publicados e/ou subscrevidos no sistema;
- rostopic type: permite visualizar o tipo de dados que um determinado tópico utiliza;
- rostopic info: mostra todas as informações referentes a um determinado tópico, tais como tipo e quais nós estão publicando e subscrevendo nesse tópico;
- rostopic pub: permite publicar mensagens (enviar dados) em um determinado tópico;
- rostopic echo: permite visualizar os dados que estão sendo publicados em um determinado tópico.

Os comandos serão executados na ordem. Em uma nova janela do terminal o seguinte comando:

rostopic list

visualizará todos os tópicos que estão rodando no sistema

/rosout

/rosout_agg

/turtle1/cmd_vel

/turtle1/color_sensor

/turtle1/pose

Esses são todos os tópicos que os nós estão publicando ou subscrevendo no momento.

O tópico /turtle1/cmd_vel é o tópico responsável por movimentar a tartaruga. O seguinte comando permite ver qual é o tipo de mensagem utilizado por esse tópico:

rostopic type /turtle1/cmd_vel

Obtendo a seguinte resposta:

geometry_msgs/Twist

Para publicar qualquer informação nesse tópico, é preciso utilizar uma mensagem do tipo geometry_msgs/Twist. Para verificar como é composto esse tipo de mensagem é utilizado o seguinte comando:

rosmsg show geometry_msgs/Twist

que retorna:

geometry_msgs/Vector3

linear: float64 x float64 y float64 z

geometry_msgs/Vector3

angular: float64 x float64 y float64 z

Será preciso, portanto, publicar mensagens que possuam seis componentes: três componentes relativos à velocidade linear e três componentes relativos à velocidade angular da tartaruga. Em geral, o comando rosmsg show pode ser usado para conhecer os detalhes de qualquer mensagem usando o seguinte comando genérico:

rosmsg show [tipo_da_mensagem].

Usando o comando rostopic info:

rostopic info /turtle1/cmd_vel

obtêm-se informações sobre o tópico /turtle1/cmd_vel:

Type: geometry_msgs/Twist

Publishers: None

Subscribers: * /turtlesim (http://xxxx-pc-055:27103/)

O comando rostopic info fornece informações referentes ao tópico. Nesse caso, o tópico /turtle1/cmd_vel é do tipo geometry_msgs/Twist, não possui nenhum nó que esteja publicando informações nesse tópico e possui um nó subscrito, que lê informações,

/turtlesim.

Para publicar mensagens manualmente para movimentar a tartaruga usando o comando rostopic pub, em uma nova janela do terminal:

rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist - - '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

Esse comando fará a tartaruga realizar um movimento circular e, depois, parar. Para entender melhor esse comando, segue uma explicação por partes:

1. rostopic pub: comando responsável por publicar mensagens em um tópico;
2. -1: indica que a mensagem será publicada apenas uma vez;
3. turtle1/cmd_vel: tópico no qual serão publicadas as mensagens;
4. geometry_msgs/Twist: tipo de mensagem que queremos publicar;

   - - : indica ao ROS que nenhum dos próximos argumentos devem ser considerados como opções do comando;
5. '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]': essa é a mensagem que será publicada, três componentes relativos à velocidade linear e três componentes relativos à velocidade angular da tartaruga, 2 vetores de 3 elementos cada. O primeiro vetor é relativo à velocidade linear e cada elemento corresponde, respectivamente, às velocidades linear em x, y e z. O segundo vetor é relativo à velocidade angular e cada elemento corresponde, respectivamente, às velocidades angulares em x, y e z. Nesse caso, a tartaruga se movimentará com velocidade linear em x igual a 2.0 e velocidade angular em z igual a 1.8.

Para que a tartaruga continue se movimentando sem parar, é preciso publicar mensagens de forma constante. Por exemplo, o turtlesim precisa receber mensagens a uma taxa de pelo menos 1 Hz para que a tartaruga se movimente sem parar. Isso pode ser facilmente alcançado com o seguinte comando:

rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist - - '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

A tartaruga irá se movimentar sem parar. O segredo é a opção -r. O número após essa opção é a frequência em Hz que desejamos publicar a mensagem, nesse caso, 1 Hz. No caso do turtlesim, aumentar a frequência de publicação da mensagem não fará a tartaruga se movimentar mais rápido, mas pode existir situações nas quais se deseja que a frequência de publicação seja mais alta. Tudo dependerá da aplicação escolhida.

Por último, para visualizar as mensagens que estão sendo publicadas em um tópico, é usado o comando rostopic echo. Publicando uma sequência de mensagens sem parar e, em uma nova janela de terminal, use o seguinte comando:

rostopic echo /turtle1/cmd_vel

Uma sequência de mensagens aparecem na janela com as informações que estão sendo publicados no tópico /turtle1/cmd_vel. Por exemplo:

linear: x: 2.0 y: 0.0 z: 0.0

angular: x: 0.0 y: 0.0 z: 1.8

linear: x: 2.0 y: 0.0 z: 0.0

angular: x: 0.0 y: 0.0 z: 1.8

Para interromper o loop sem fim, é usada a combinação de teclas Ctrl+C na janela de terminal para finalizar a execução do comando.

Depois de publicar uma sequência de mensagens, podemos utilizar novamente o comando rostopic info para observar que agora temos um novo nó publicando no tópico /turtle1/cmd_vel:

rostopic info /turtle1/cmd_vel

retornando:

Type: geometry_msgs/Twist

Publishers:

* /rostopic_6016_1463934543457 (http://ascc-pc-035:53187/)

Subscribers:

* /turtlesim (http://ascc-pc-035:53084/)

APÊNDICE III – ASSISTENTE DE CONFIGURAÇÃO DO MOVEIT! (EXEMPLO COM ROBÔ PR2)

O primeiro grande robô da Willow Garage é o PR2. Ele tem um tamanho um pouco menor que um humano. O PR2 foi projetado para ser uma plataforma de hardware e software comum entre pesquisadores. O PR2 é uma evolução do PR1, uma plataforma robótica desenvolvida na Universidade de Stanford. A sigla PR significa "personal robot"(robô pessoal).

Iniciar o MoveIt! Assistente de Configuração

Em um terminal:

roslaunch moveit_setup_assistant setup_assistant.launch

Isso trará a tela inicial com duas opções: Create New MoveIt! Configuration Package ou Edit Existing MoveIt! Configuration Package.

Clicando no botão Create New MoveIt! Configuration Package será exibida a seguinte tela:

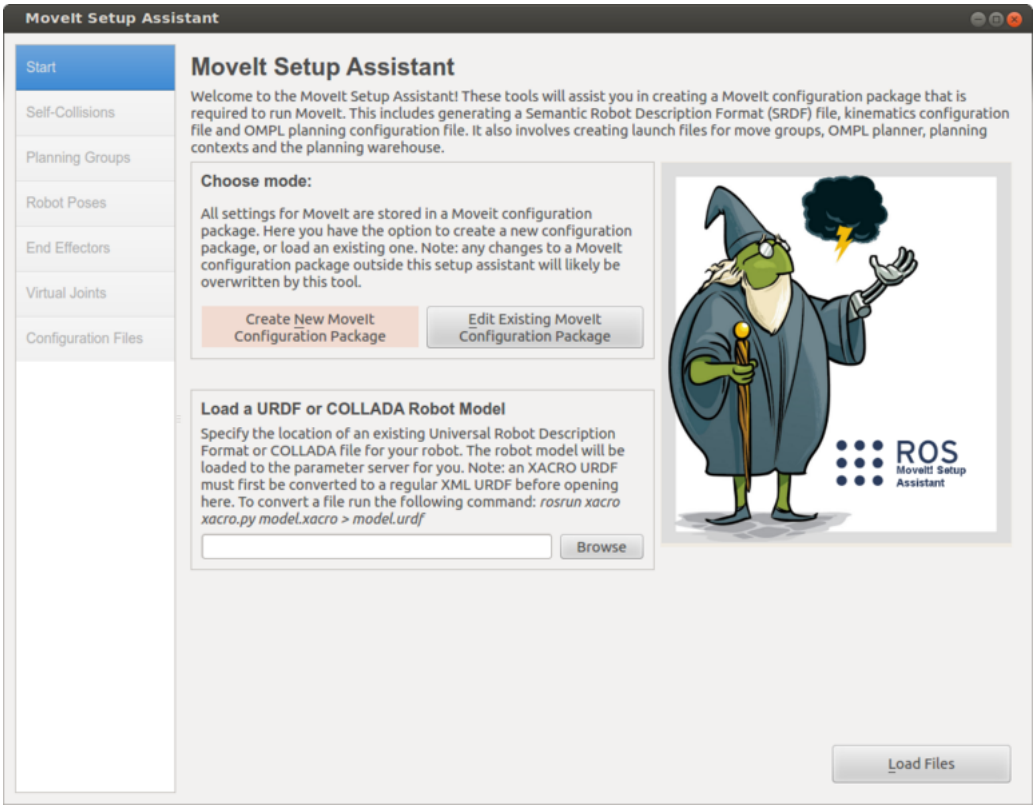


Figura III.1: Assistente de configuração do MoveIt! 1/2 Fonte: Moveit2015

Clicando no botão de navegação e navegando até o arquivo pr2.urdf.xacro instalado quando foi instalado o ros-indigo-moveit-full-pr2. (Este arquivo é instalado em /opt/-ros/indigo/share/pr2_description/robots/pr2.urdf.xacro no Ubuntu com o ROS Indigo.) Escolhendo o arquivo e clicando em Carregar Arquivos, o Assistente de Configuração carregará os arquivos apresentando a tela seguinte:

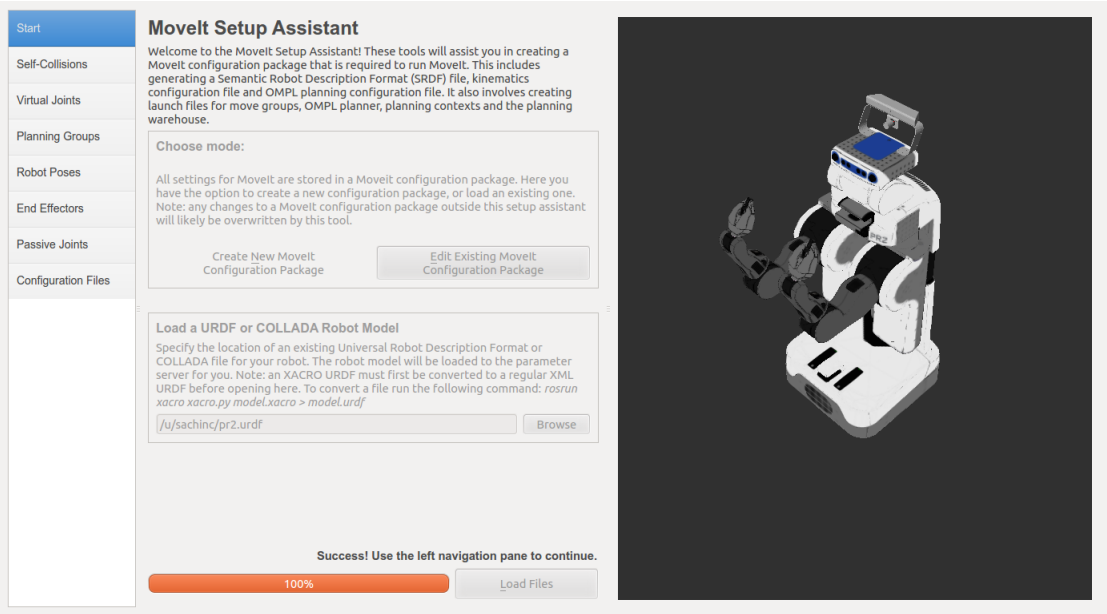


Figura III.2: Assistente de configuração do MoveIt! 2/2 Fonte: Moveit2015

Matriz de auto-colisão

Clicando em Self-Collisions no lado esquerdo e, a seguir, no botão Regenerate Default Collision Matrix, o Assistente de Configuração apresentará os resultados de seu cálculo na tabela principal(Figura III.4).

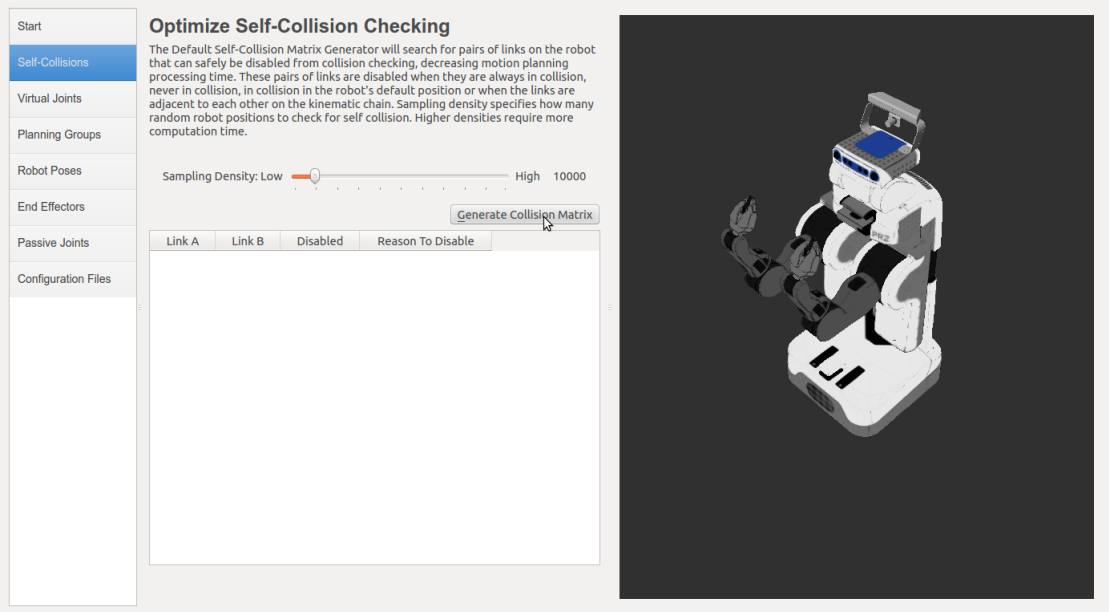


Figura III.3: Auto-colisão 1/2 Fonte: Moveit2015

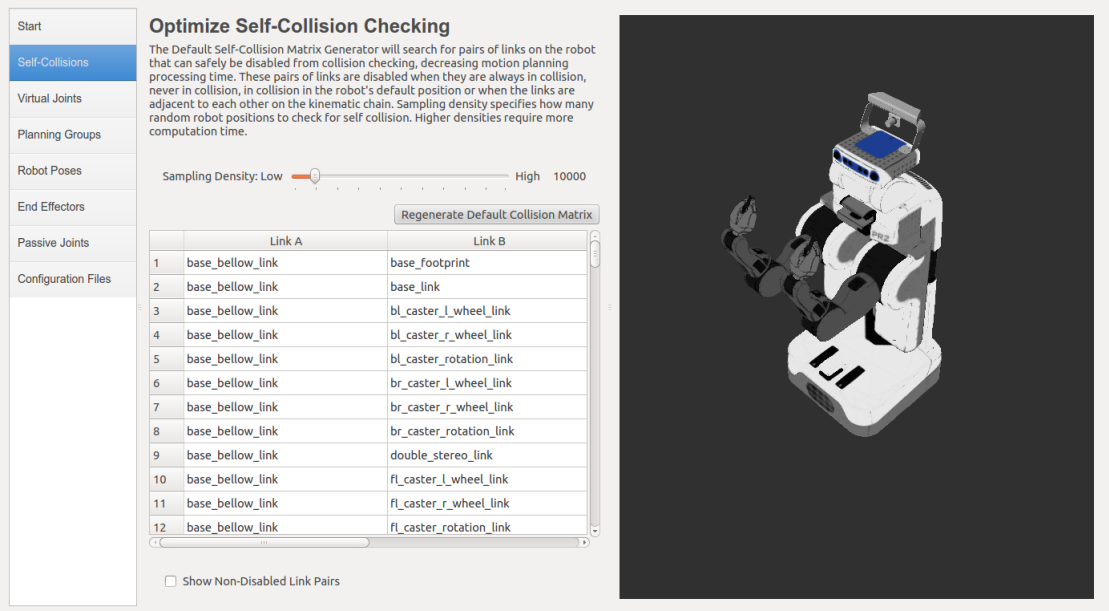


Figura III.4: Auto-colisão 2/2 Fonte: Moveit2015

Adicionar articulações virtuais

As articulações virtuais são usadas principalmente para conectar o robô ao mundo. Para o PR2, define-se apenas uma junta virtual anexando base_footprint do PR2 à posição no mundo odom_combined. Essa articulação virtual representa o movimento da base do robô em um plano.

Sequência:

Clicar no seletor do painel Virtual Joints e depois em Add Virtual Joint

- Definir virtual joint name como "virtual_joint"
- Definir o child link como "base_footprint"e o parent frame name como ?odom_combined?.
- Definir the Joint Type como "planar".
- Clicar em Save, aparecerá a tela da Fugura III.5

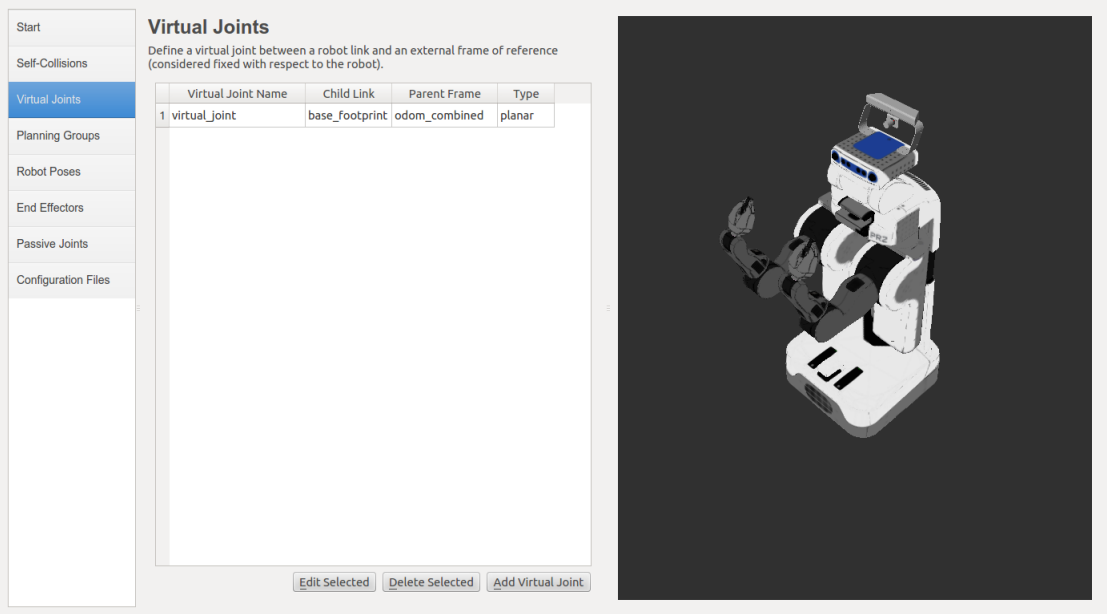


Figura III.5: Articulações virtuais Fonte: Moveit2015

Adicionar grupos de planejamento

Os grupos de planejamento descrevem semanticamente partes diferentes do robô.

- Clicar em Planning Groups.
- Clicar em Add Group, aparecerá a tela da Figura III.6

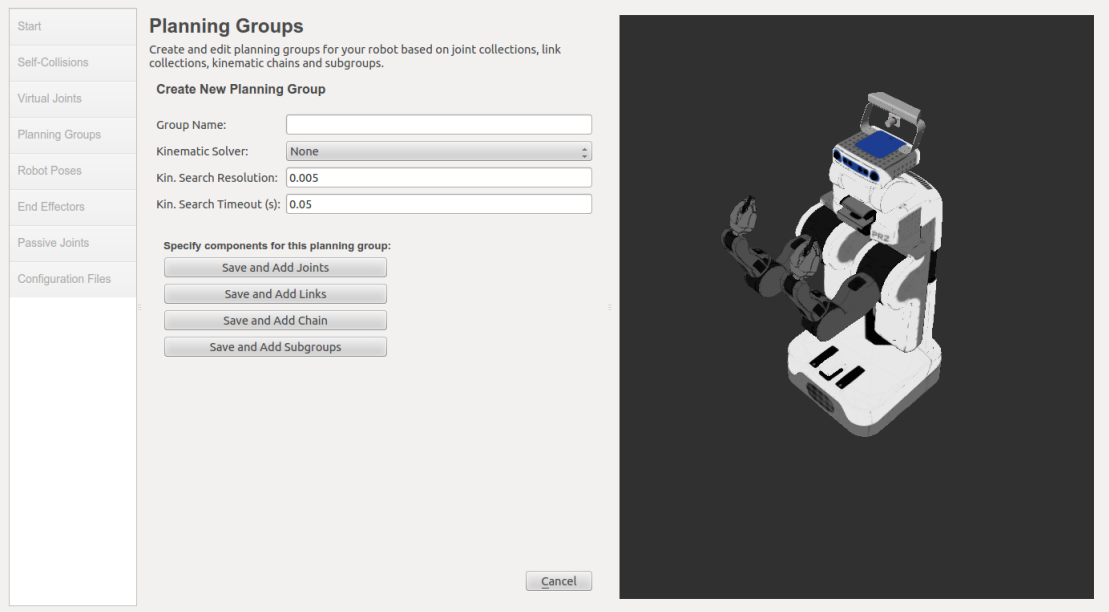


Figura III.6: Grupos de planejamento 1/5 Fonte: Moveit2015

- Definir Group Name como right_arm
- Definir kdl_kinematics_plugin/KDLKinematicsPlugin como kinematics solver.
- Deixar Kin. Search Resolution e Kin. Search Timeout nos valores padrão(Figura III.7).

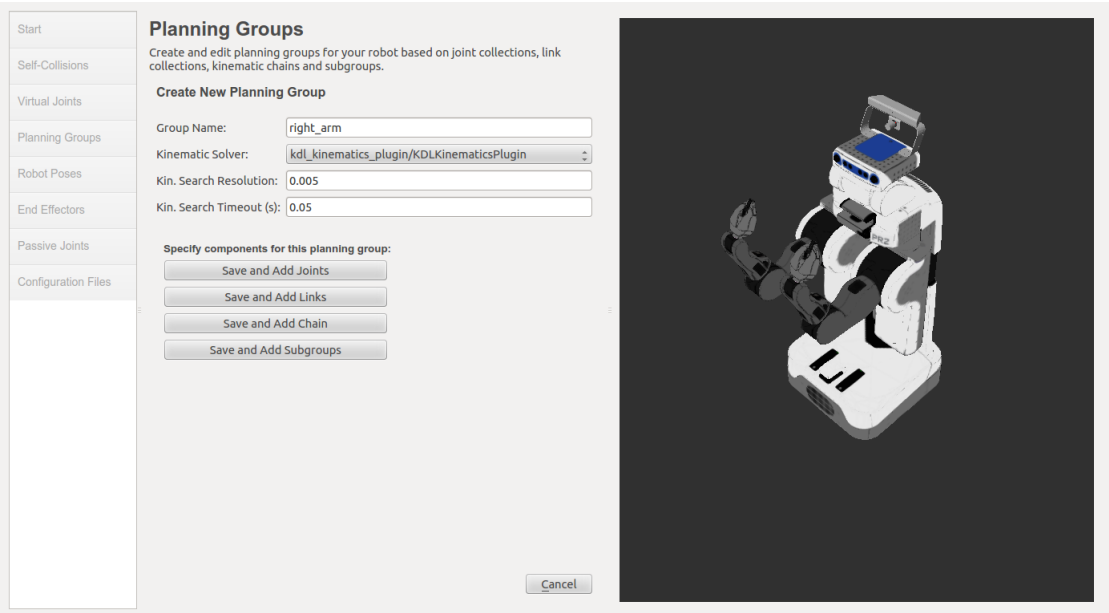


Figura III.7: Grupos de planejamento 2/5 Fonte: Moveit2015

Clicando em Save and Add Joint, aparecerá uma lista de articulações no lado esquerdo. Agora, para configurar o right_arm, selecionar as articulações de r_shoulder_pan_joint até r_wrist_roll_joint e adicionar à lista Selected Links no lado direito. Depois de salvar, o resultado será similar à tela da Figura III.8

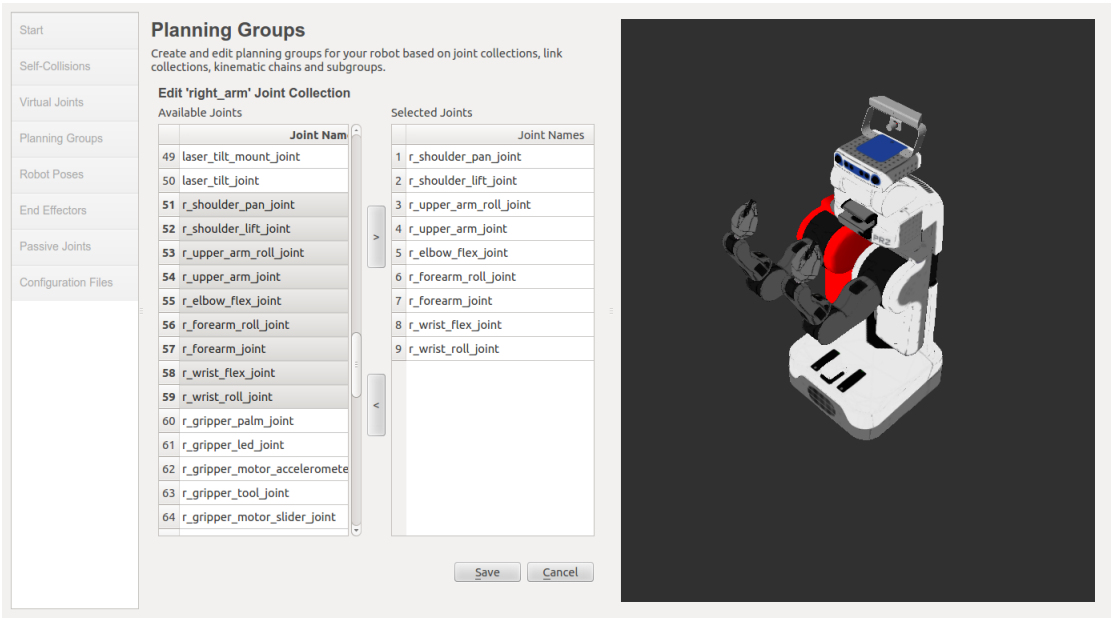


Figura III.8: Grupos de planejamento 3/5 Fonte: Moveit2015

depois de clicar em save, aparecerá a tela da Figura III.9

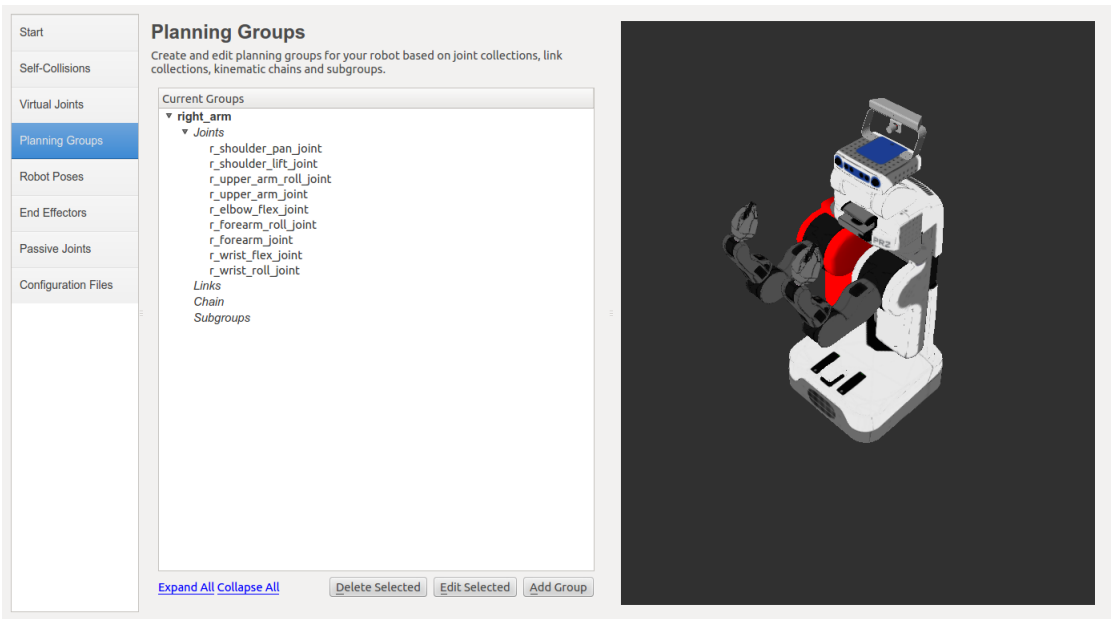


Figura III.9: Grupos de planejamento 4/5 Fonte: Moveit2015

pode se repetir o mesmo processo para o braço esquerdo left_arm, esta vez escolhendo as articulações de l_shoulder_pan_joint até l_wrist_roll_joint

Adicionar as garras

Para adicionar dois grupos para as garras direitas e esquerdas o procedimento será um pouco diferente.

- Clicar em Add Group.
- Definir Group Name como right_gripper
- Deixar Kin. Search Resolution e Kin. Search Timeout nos valores padrão.
- Clicar em the Save and Add Links.
- Selecionar todas as ligações right_gripper* e adicionar à lista Selected Links no lado direito.
- Salvar
- Repetir o procedimento para left_gripper selecionando, desta vez, as ligações left_gripper no lugar de right_gripper.

A telaa da Figura III.10 mostra a nova situação:

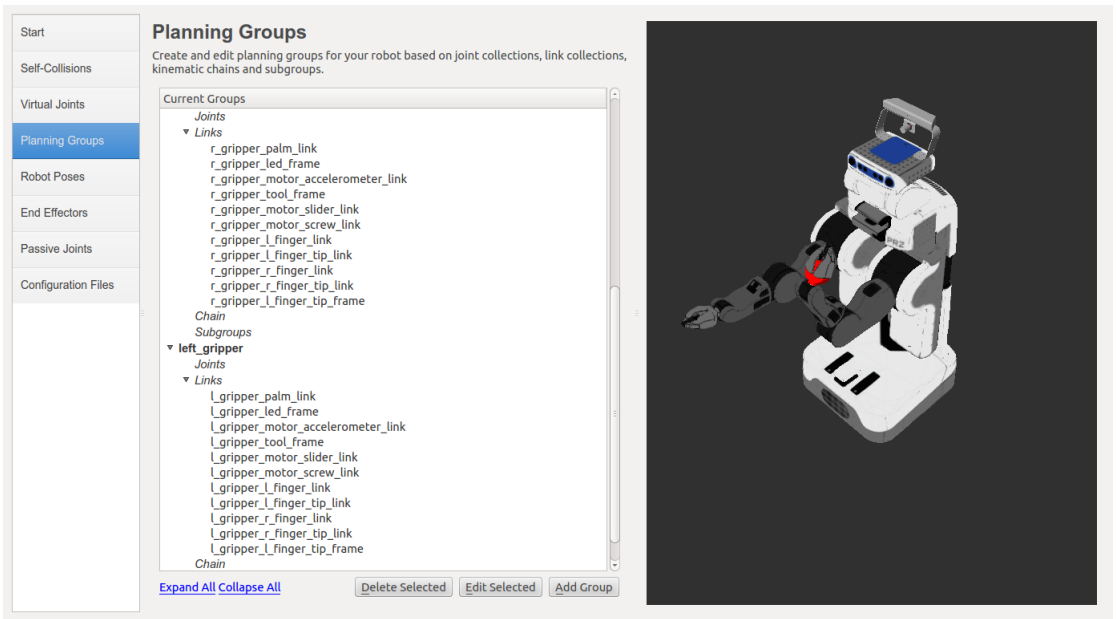


Figura III.10: Grupos de planejamento 5/5 Fonte: Moveit2015

Adicionar poses do robô

O Assistente de Configuração permite adicionar certas poses fixas na configuração. Isso ajuda se, por exemplo, precisa definir uma determinada posição do robô como uma posição inicial(FiguraIII.11).

- Clicar em Robot Poses
- Clicar em Adicionar Pose e escolher um nome para a pose. O robô estará em sua posição Default, os valores poderão ser alterados.

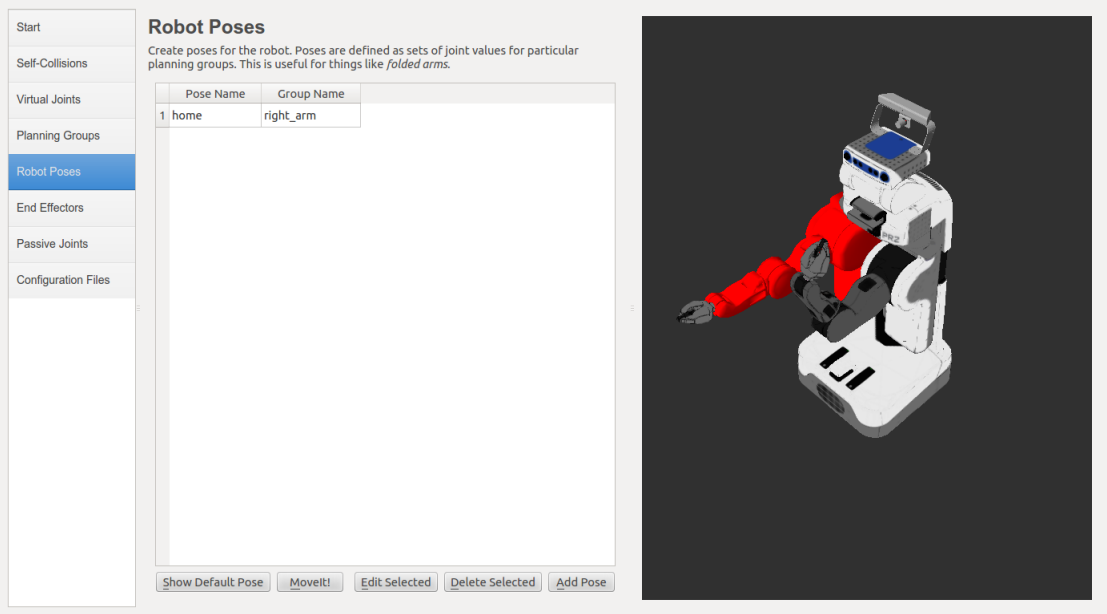


Figura III.11: Poses do robô Fonte: Moveit2015

End Effectors

Na robótica, um end effector é o dispositivo no final de um braço robótico, projetado para interagir com o ambiente. A natureza exata deste dispositivo depende da aplicação do robô.Uma vez definidas as garras direita e esquerda do PR2 podem ser definidos dois grupos especiais: os end effectors. A designação desses grupos como effectors permite que algumas operações especiais ocorram internamente.

- Clicar em End Effectors.
- Clicar em Add End Effectors.

- Definir right_eef como End Effector Name para a garra direita.
- Selecionar right_gripper como End Effector Group.
- Selecionar r_wrist_roll_link como Parent Link para este end-effector.
- Deixar Parent Link em branco.

A tela da FiguraIII.12 mostra as definições:

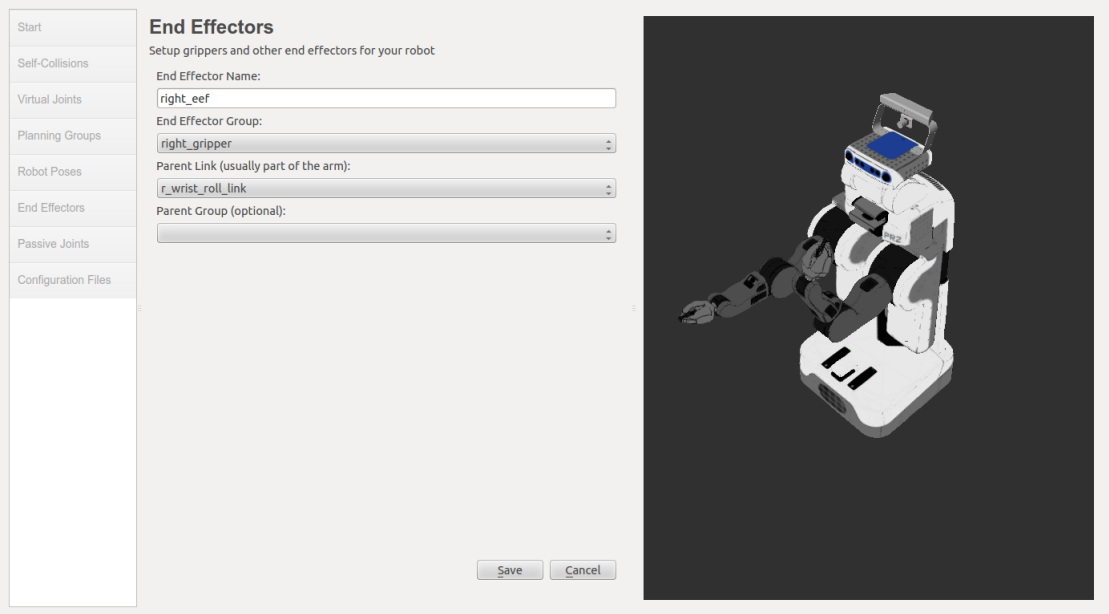


Figura III.12: End effectors Fonte: Moveit2015

- Salvar.
- Adicionar o left_eef de maneira semelhante.

Adicionar junções passivas

O PR2 não possui juntas passivas, portanto esse passo não será definido. **Geração dos arquivos de configuração**

Um último passo, a geração de todos os arquivos de configuração necessários para começar a usar o MoveIt!

- Clicar no painel Configuration File.

- Escolher um local e um nome para o pacote ROS que será gerado contendo o novo conjunto de arquivos de configuração.

- Clicar no botão Generate Package. O Assistente de Configuração agora gerará e gravará um conjunto de arquivos de inicialização e configuração no diretório escolhido. Todos os arquivos gerados aparecerão na guia Generated Files/Folders e, clicando em cada um deles se obterá uma descrição do que eles contêm(Figura III.13).

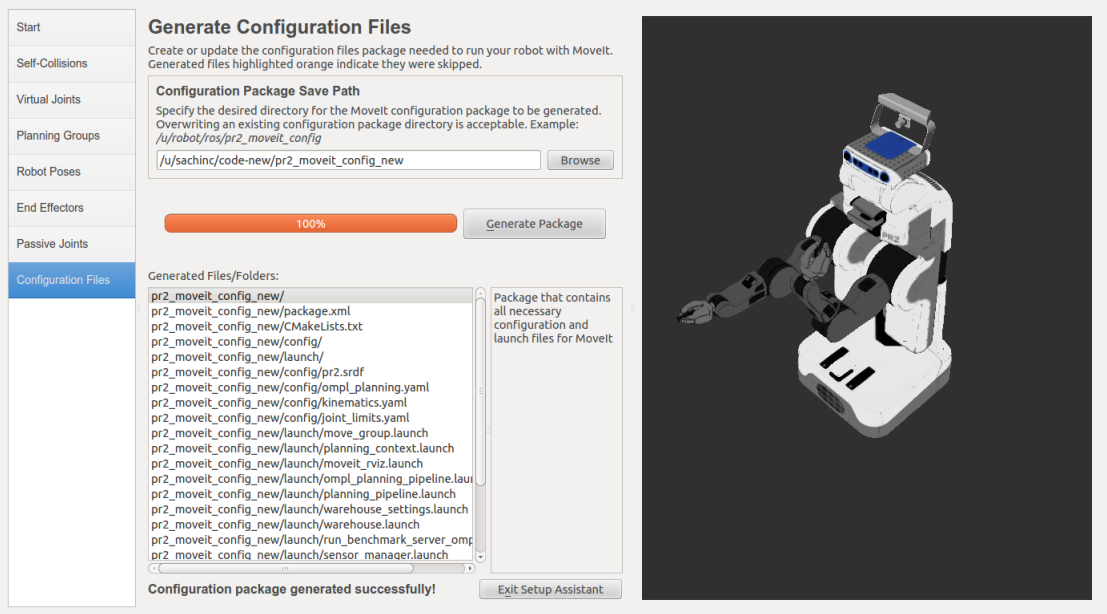


Figura III.13: Arquivos de configuração Fonte: Moveit2015

Uma vez completada a configuração será possível manipular o robô PR2.

APÊNDICE IV – TUTORIAL PARA MANIPULAR O BAXTER

Criação área de trabalho de desenvolvimento do Baxter

Braço do Baxter

Na Figura IV.1 é mostrado o braço do Baxter com relativas juntas.

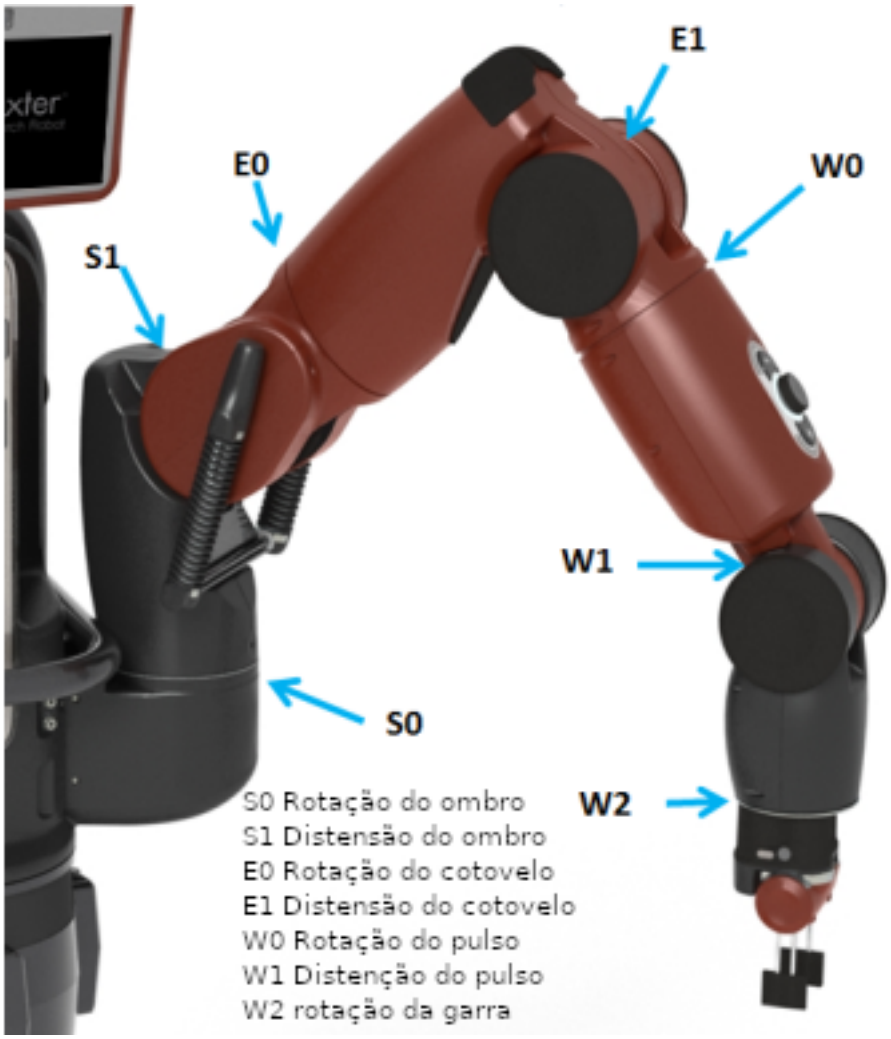


Figura IV.1: Braço do Baxter Fonte: BAXTER2018

Considerando o uso do ros Indigo:

mkdir -p /ros_ws/src

source /opt/ros/indigo/setup.bash

cd /ros_ws

catkin_make

catkin_make install

Instalar as dependências do SDK (Software development kit)

sudo apt-get update

sudo apt-get install git-core python-argparse python-wstool python-vcstools python-rosdep ros-indigo-control-msgs ros-indigo-joystick-drivers

Instalar Baxter Research Robot SDK

cd /ros_ws/src

wstool init .

wstool merge https://raw.githubusercontent.com/RethinkRobotics/baxter/master/baxter_sdk.rosinstall

wstool update

source /opt/ros/indigo/setup.bash

cd /ros_ws

catkin_make

catkin_make install

Configurar a comunicação Baxter/área de trabalho do ROS

wget https://github.com/RethinkRobotics/baxter/raw/master/baxter.sh

chmod u+x baxter.sh

cd /ros_ws

gedit baxter.sh

No arquivo baxter.sh editar o campo *"baxter_hostname"*:

baxter_hostname="localhost"

editar o campo *"your_ip"*:

your_ip="192.168.XXX.XXX"

editar o campo *"ros_version"*:

ros_version="indigo"

salvar e fechar o script *baxter.sh*

Verificar o ambiente

env | grep ROS

o resultado poderá ser parecido com o seguinte:

ROS_ROOT=/opt/ros/indigo/share/ros

ROS_PACKAGE_PATH=/home/<usuario>/ros_ws/src:/opt/ros/indigo/share

ROS_MASTER_URI=http://localhost:11311

ROS_VERSION=1

ROSLISP_PACKAGE_DIRECTORIES=/home/<usuario>/ros_ws/devel/share/common-lisp

__ROS_PROMPT=1

ROS_DISTRO=indigo

ROS_IP=<o_seu_ip>

ROS_ETC_DIR=/opt/ros/indigo/etc/ros

Exemplo de manipulação interativa do Baxter

Simples aplicação interativa onde o robô Baxter movimenta os braços para simular uma saudação

Iniciar python: python

importar os módulos necessários:

rospy - ROS Python API:

>>> import rospy

baxter_interface - Baxter Python API:

>>> import baxter_interface

inicializar um nó ROS:

>>> rospy.init_node('Hello_Baxter')

criar uma instância da classe Limb da baxter_interface:

>>> limb = baxter_interface.Limb('right')

obter os ângulos das articulações do braço direito:

>>> angles = limb.joint_angles()

imprimir os valores dos ângulos atuais:

>>> print angles

redefinir os ângulos:

>>> angles['right_s0']=0.0

>>> angles['right_s1']=0.0

>>> angles['right_e0']=0.0

>>> angles['right_e1']=0.0

>>> angles['right_w0']=0.0

>>> angles['right_w1']=0.0

>>> angles['right_w2']=0.0

imprimir os valores dos ângulos redefinitos:

>>> print angles

Posição inicial (Figura IV.2)

mover o braço direito para a posição definida:

>>> limb.move_to_joint_positions(angles)

atribuir duas posições aos braços para simular uma saudação:

primeira posição:

>>> wave_1 = {'right_s0': -0.459, 'right_s1': -0.202, 'right_e0': 1.807, 'right_e1': 1.714, 'right_w0': -0.906, 'right_w1': -1.545, 'right_w2': -0.276}

segunda posição:

>>> wave_2 = {'right_s0': -0.395, 'right_s1': -0.202, 'right_e0': 1.831, 'right_e1':

1.981, ’right_w0’: -1.979, ’right_w1’: -1.100, ’right_w2’: -0.448}

ciclo for para repetir a saudação:

>>> for _move in range(3):

...

limb.move_to_joint_positions(wave_1)

Posição 1 (Figura IV.3)

...

limb.move_to_joint_positions(wave_2)

Posição 2 (Figura IV.4)

encerrar python

>>> quit()

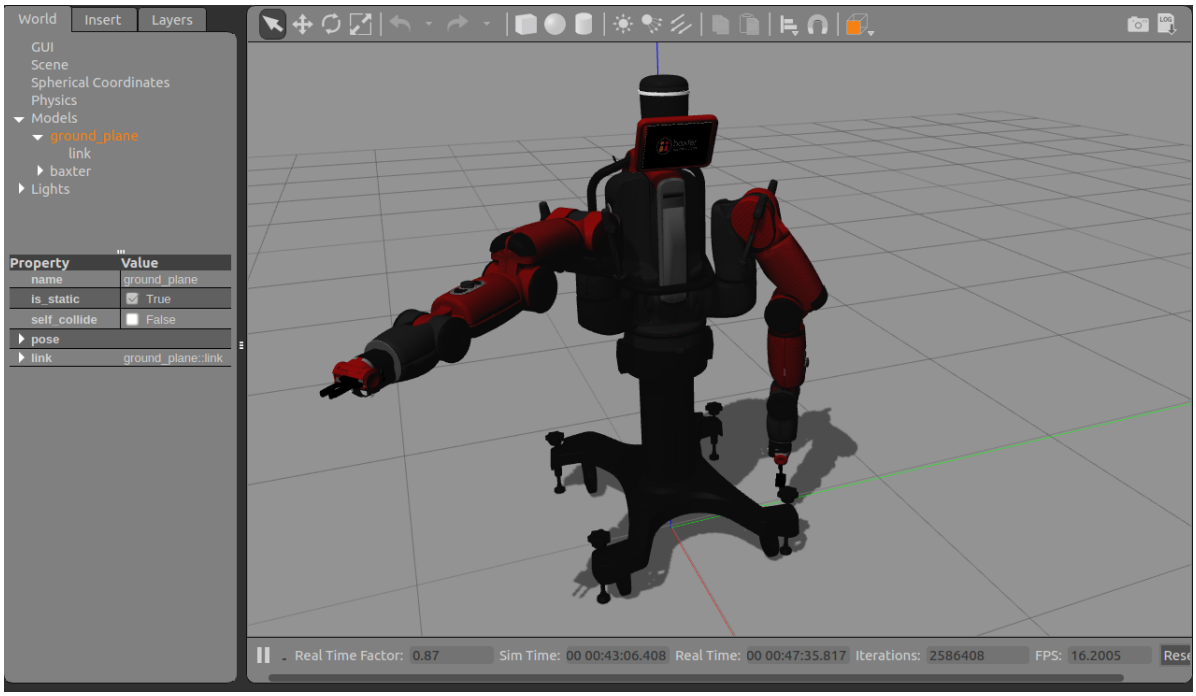


Figura IV.2: Posição inicial Fonte: Elaborado pelo autor

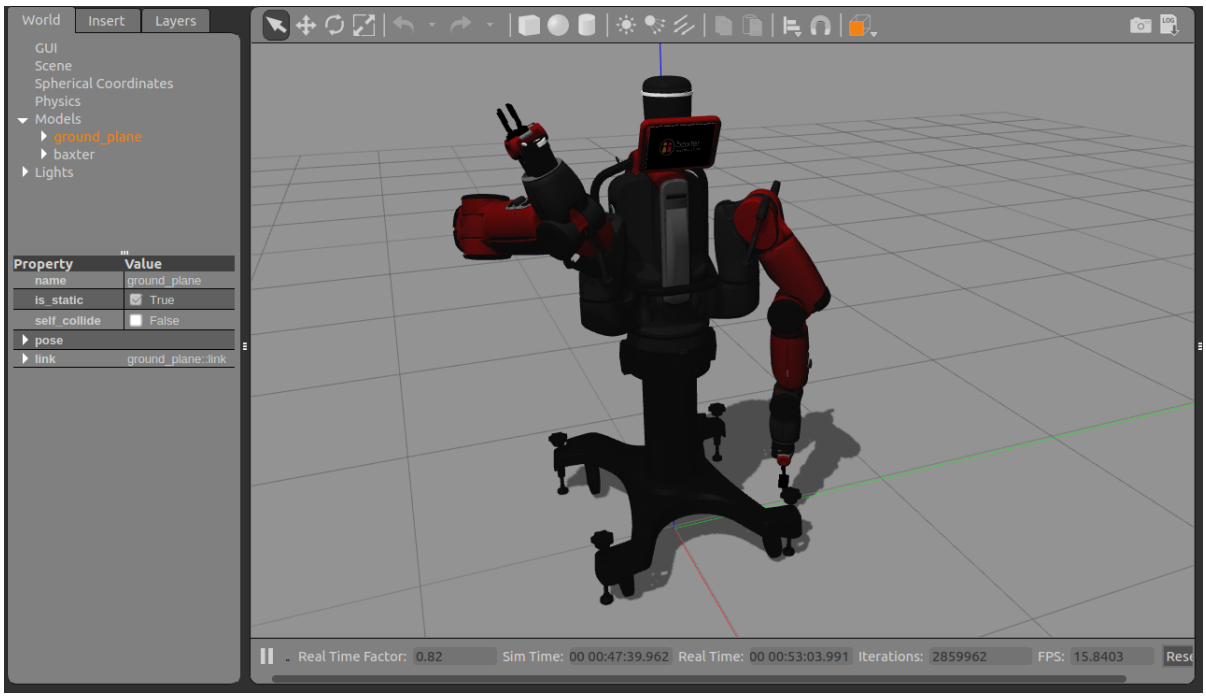


Figura IV.3: Posição 1 Fonte: Elaborado pelo autor

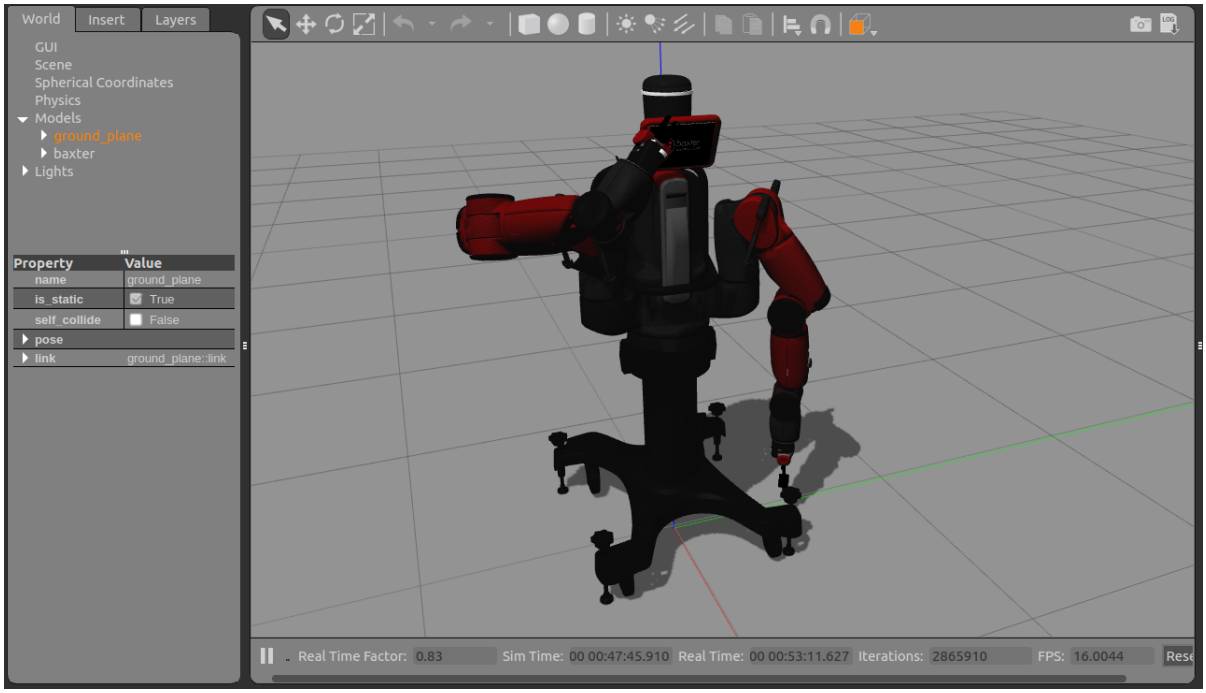


Figura IV.4: Posição 2 Fonte: Elaborado pelo autor

APÊNDICE V – TRABALHANDO COM O BAXTER UTILIZANDO MOVEIT! E RVIZ

Preparação do ambiente de trabalho

Considerando um espaço de trabalho ws_ros e a distribuição ROS Indigo vamos configuraçãr Baxter MoveIt na área de trabalho do ROS

cd ~/ros_ws/src

git clone https://github.com/ros-planning/moveit_robots.git

atualizar os pacotes:

sudo apt-get update

Instalando o moveit!:

sudo apt-get install ros-indigo-moveit-full

Executando o catkin_make para fazer as novas adições no espaço de trabalho do ROS

cd /ros_ws/

./baxter.sh

catkin_make

MoveIt!fornece recursos incluindo Cinemática (IK - Inverse Kinematics, FK - Forward Kinematics, Jacobiana), Planejamento de Movimento (OMPL - Open Motion Planning Library, SBPL - Search-based planning library, CHOMP - Covariant Hamiltonian Optimization for Motion Planning) integrados como plugins, Environment Representation (representação de robôs, representação de ambiente, checagem de colisão, avaliação de restrições), execução usando benchmarking, banco de dados warehouse para armazenamento (cenas, estados de robôs, planos de movimento), API C ++ / Python.

Tutorial de execução

Baxter no Gazebo:

roslaunch baxter_gazebo baxter_world.launch

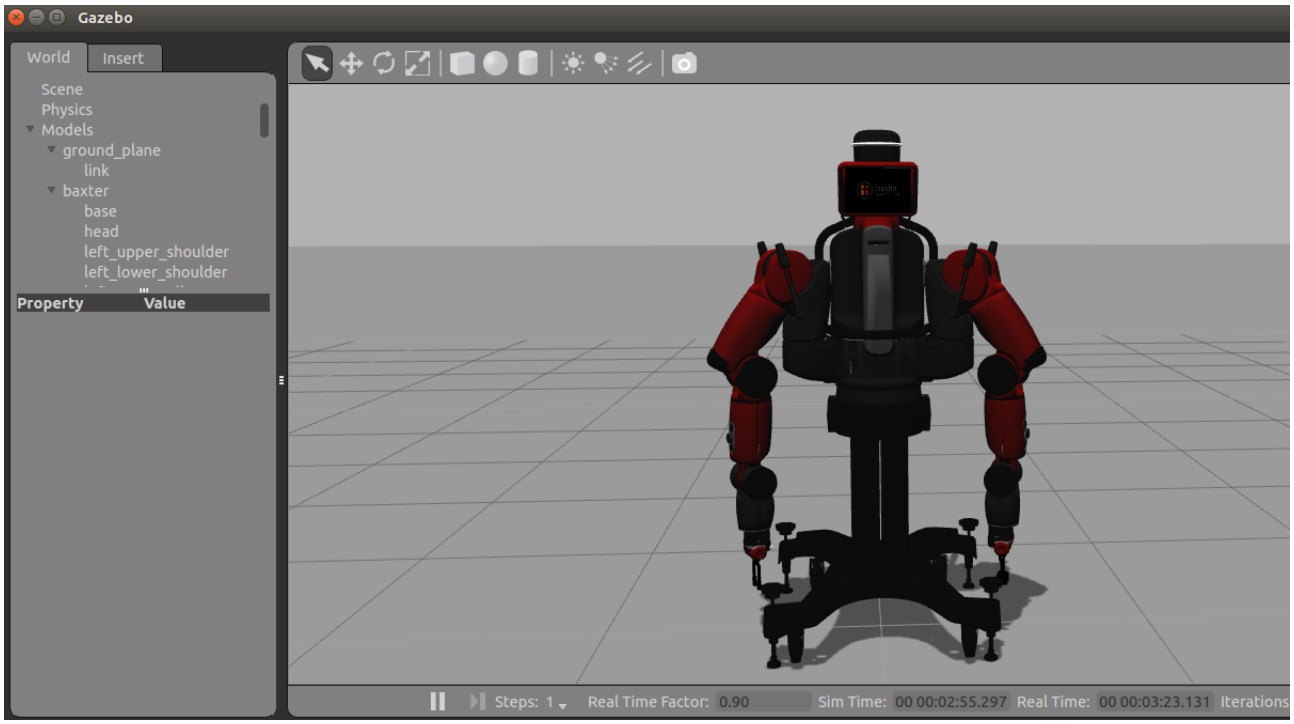


Figura V.1: O mundo de Baxter Fonte: BAXTER2018

Habilitar o Baxter:

rosrun baxter_tools enable_robot.py -e

Posicionar os braços no estado "abertos"

rosrun baxter_tools tuck_arms.py - u

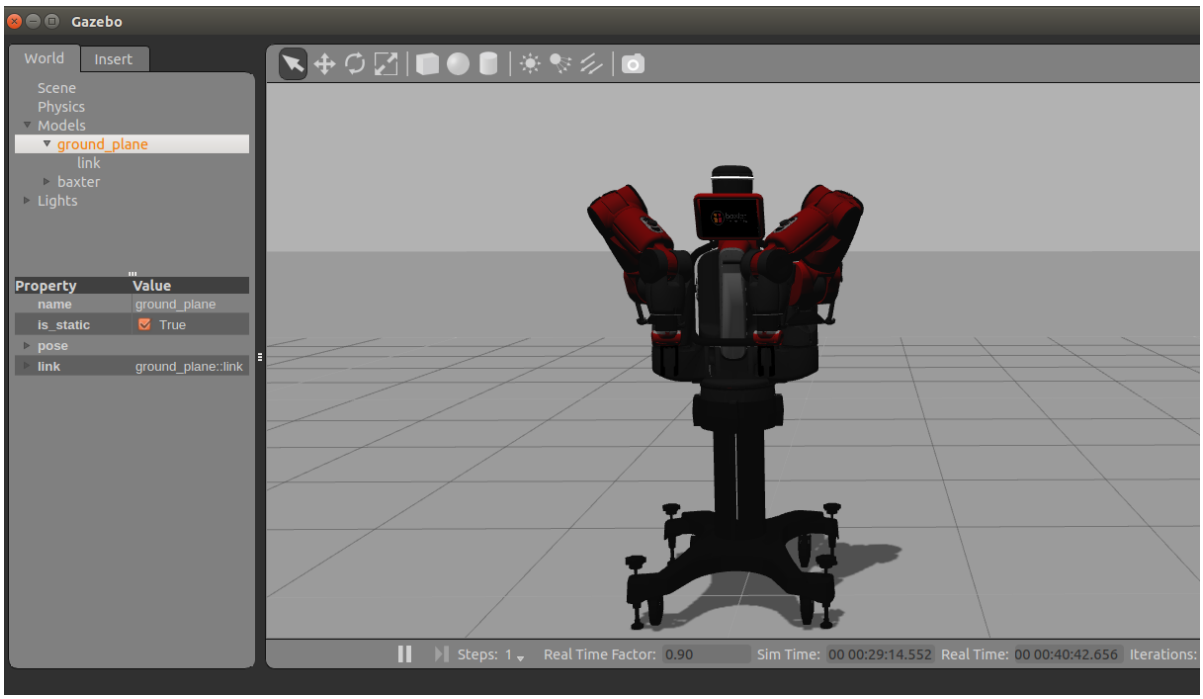


Figura V.2: Baxter na posição "untuck"Fonte: BAXTER2018

Inicializar o controle de trajetória:

rosrun baxter_interface joint_trajectory_action_server.py

Lançar o MoveIt! RViz Plugin:

roslaunch baxter_moveit_config demo_baxter.launch

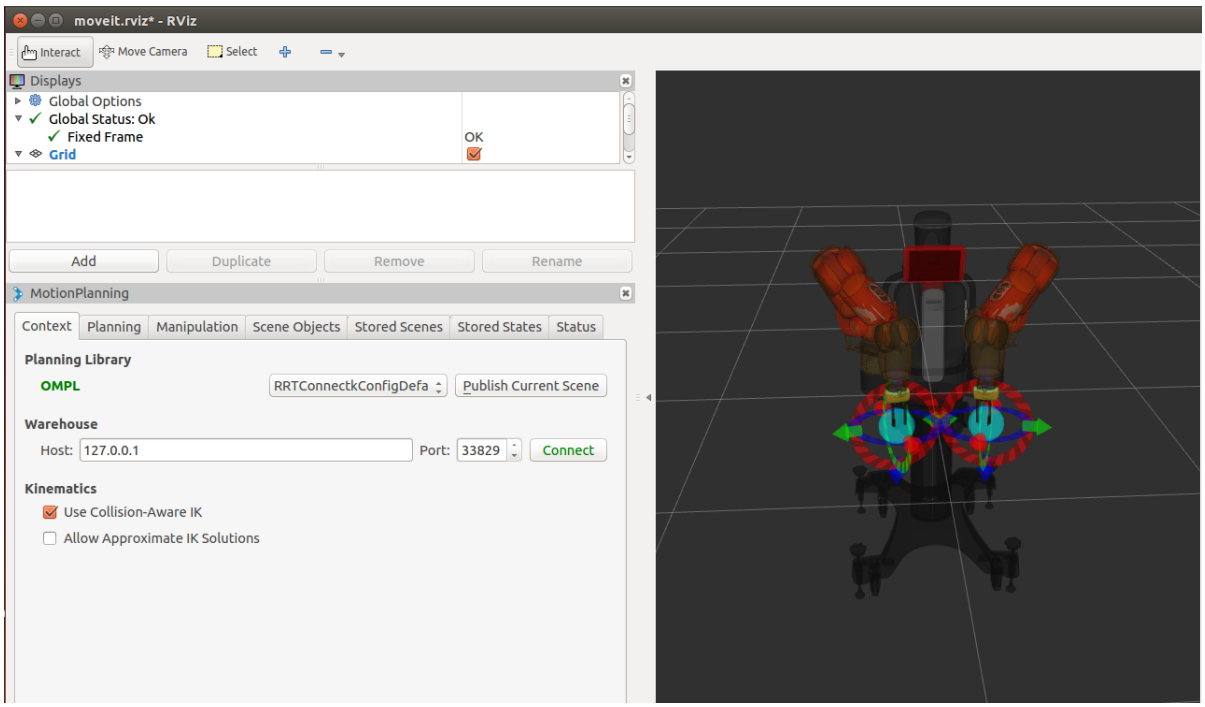


Figura V.3: Janela com painéis de exibição e planejamento de movimento Fonte: BAXTER2018

Na Figura V.3, as janelas Displays e Motion Planning são mostradas à esquerda com as informações da guia Context exibidas. À direita é simulado Baxter na posição inicial.

Janela MotionPlanning:

Context	Publicar cena atual e salvar a cena em um banco de dados.
Planning	Define o estado inicial, o estado do objetivo e planeja e executa movimentos dos braços do Baxter.
Scene Object	Importa ou exporta objetos como figuras geometricas 3D ou mesas de um arquivo em disco.
Stored Scene	Armazena cenas em um banco de dados.
Stored States	Armazena e carrega estados do robô.
Status	Status

Figura V.4: Janela MotionPlanning Fonte: BAXTER2018

Planejar um movimento

- Definir a posição inicial: Clicar em Planning, em Query escolher Select Start State, depois "current"e clicar em Update.

- Definir a posição final: Usando setas e anéis movimentar os braços do baxter para a posição final.

A Figura V.5 mostra as posições inicial e final respetivamente em vermelho e amarelo:

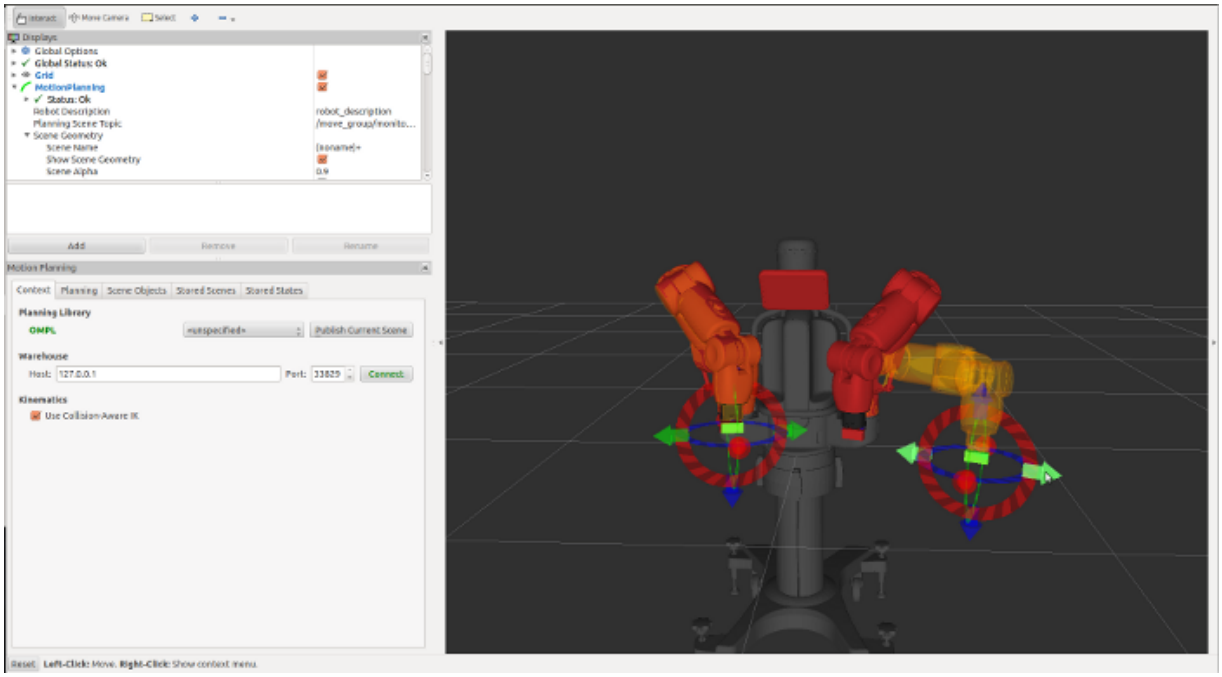


Figura V.5: Planejamento de trajetória Fonte: BAXTER2018

Para planejar e executar a trajetória entre estado inicial e final selecionar (Figura V.6):

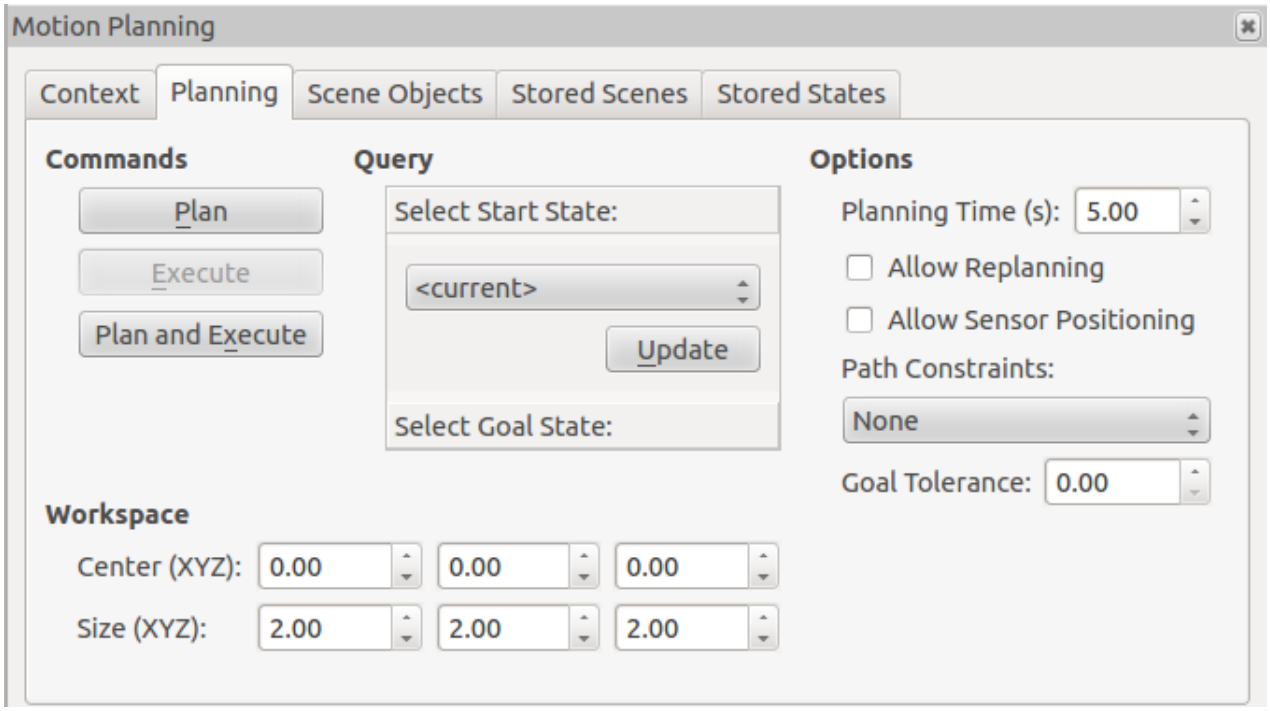


Figura V.6: Planning in Motion Planning Fonte: BAXTER2018

Clicando em Plan será visualizado o caminho planejado e clicando em Execute o mesmo será executado(Figura V.7).

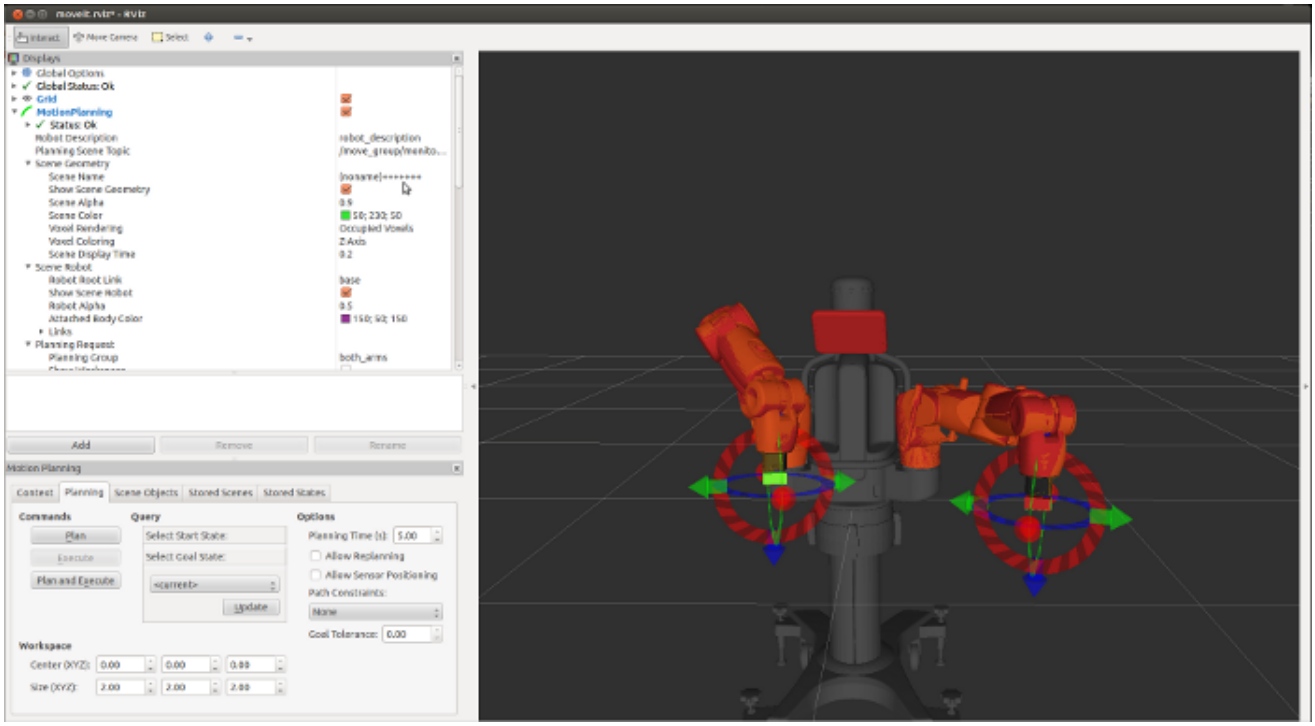


Figura V.7: Execução da trajetória Fonte: BAXTER2018

Introduzindo um elemento no ambiente

Selecionar *Scene Object* em *Motion Planning* (Figura V.8):

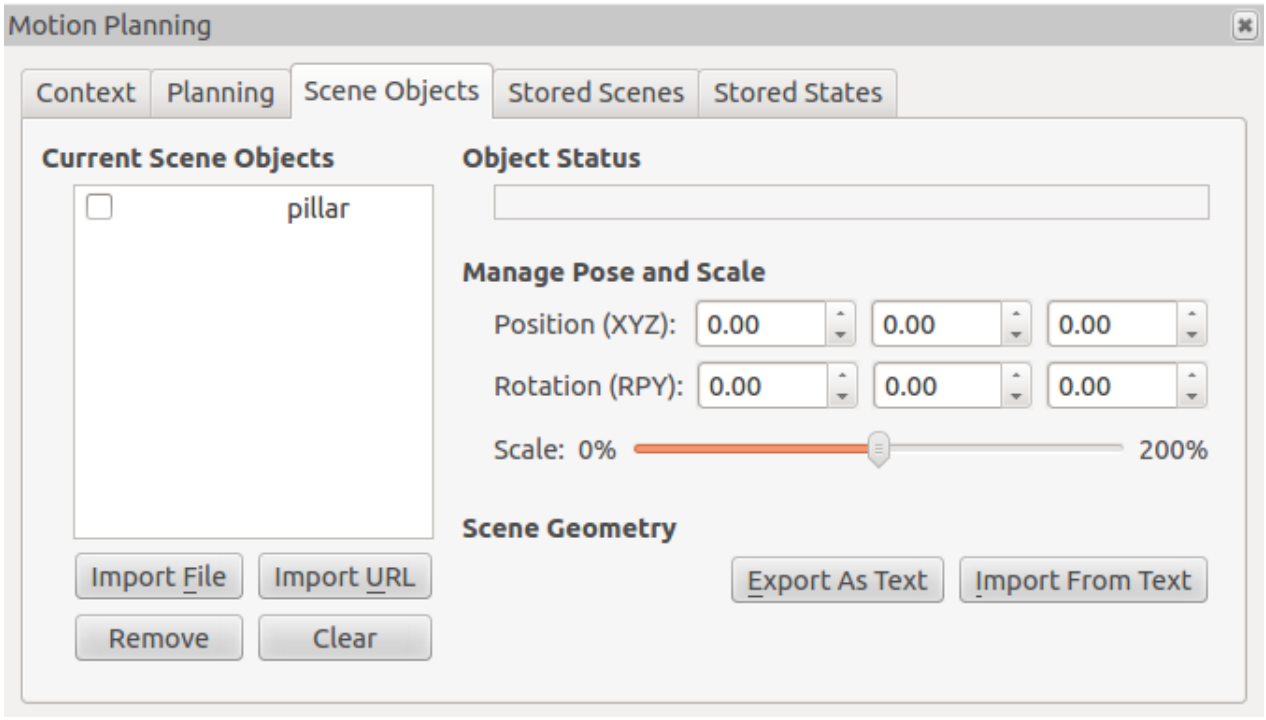


Figura V.8: Scene Objects in Motion Planning Fonte: BAXTER2018

Criar um objeto em um arquivo de texto para ser importado para o ambiente.

Em um terminal:

roscd baxter_moveit_config

mkdir baxter_scenes

gedit baxter_scenes/baxter_objectscene

copiar a seguinte definição do objeto:

object

* object

1

box

0.2 0.2 1

0.6 0.15 0

0 0 0 1

0 0 0 0 .

Salvar e sair.

Importar a cena selecionando *Import From Tex* em *Scene Geometry* (Figura V.9)

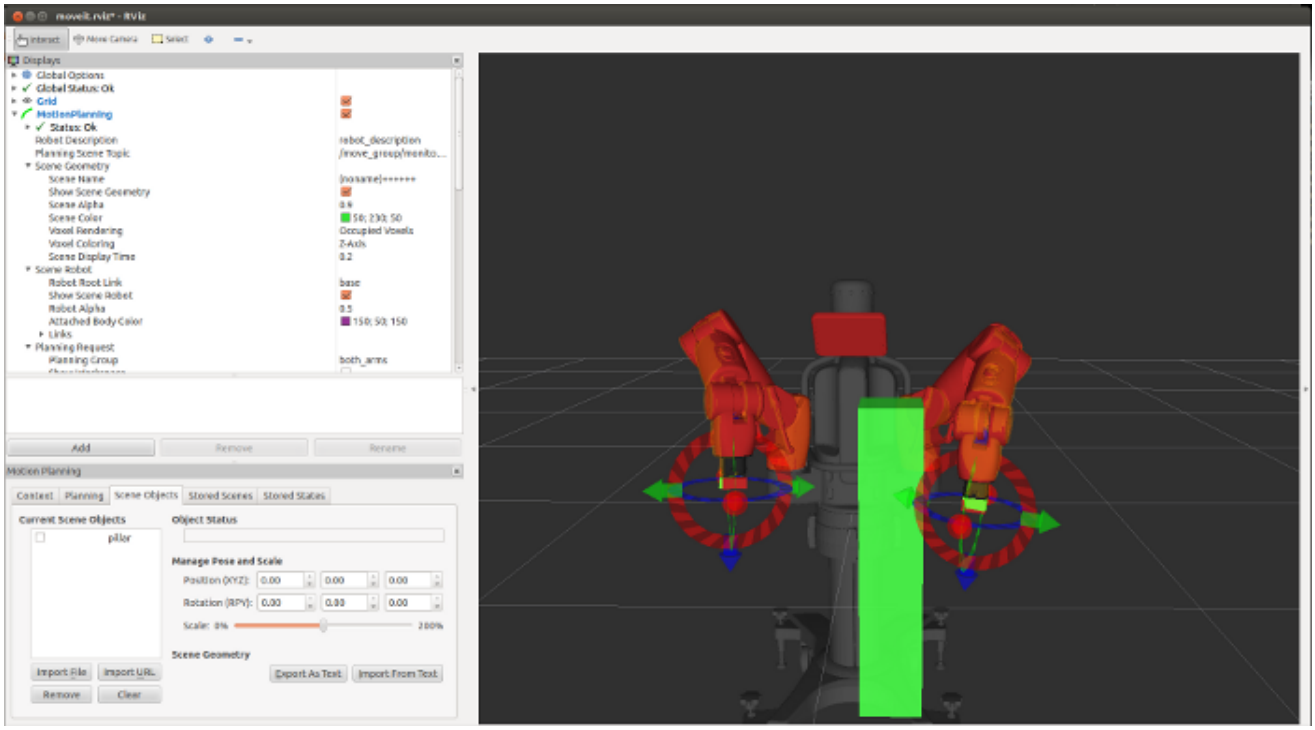


Figura V.9: Nova cena com objeto Fonte: BAXTER2018

A cena deve ser publicada para ser reconhecida clicando em *Publish Current Scene*

(Figura V10):

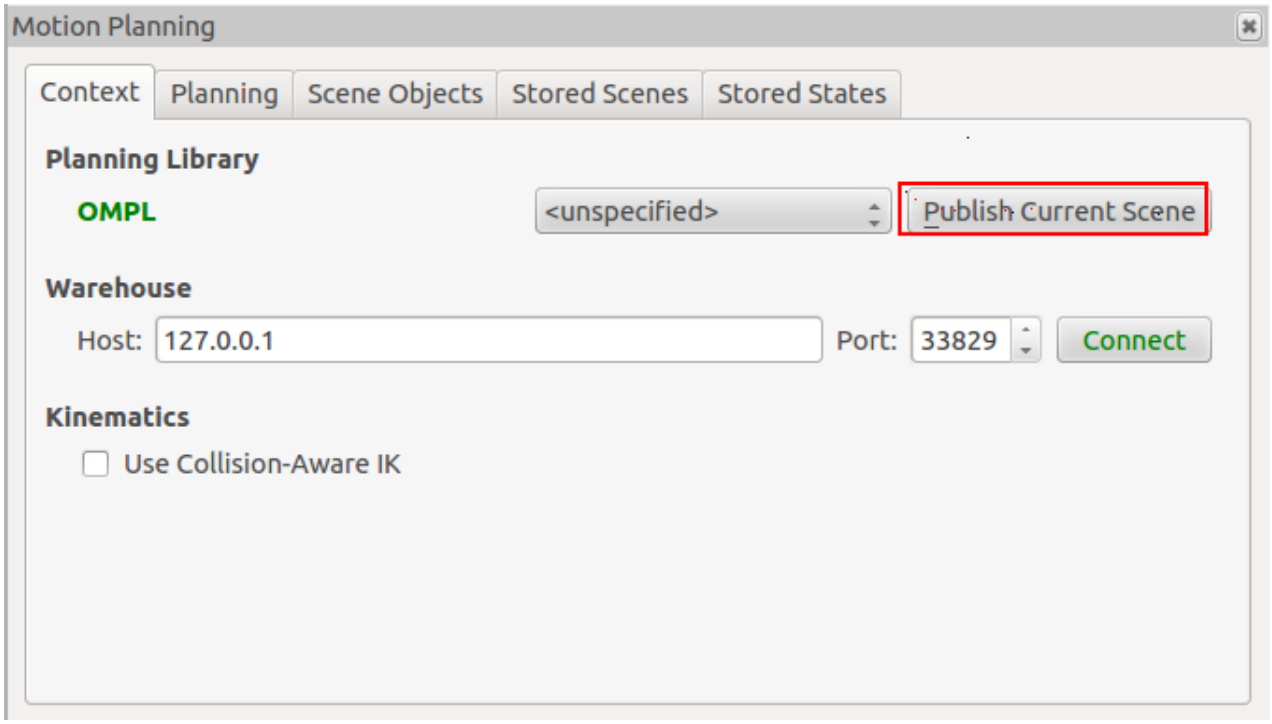


Figura V.10: Publicar cena Fonte: BAXTER2018

Definir a posição final do lado oposto do objeto (Figura V.11).

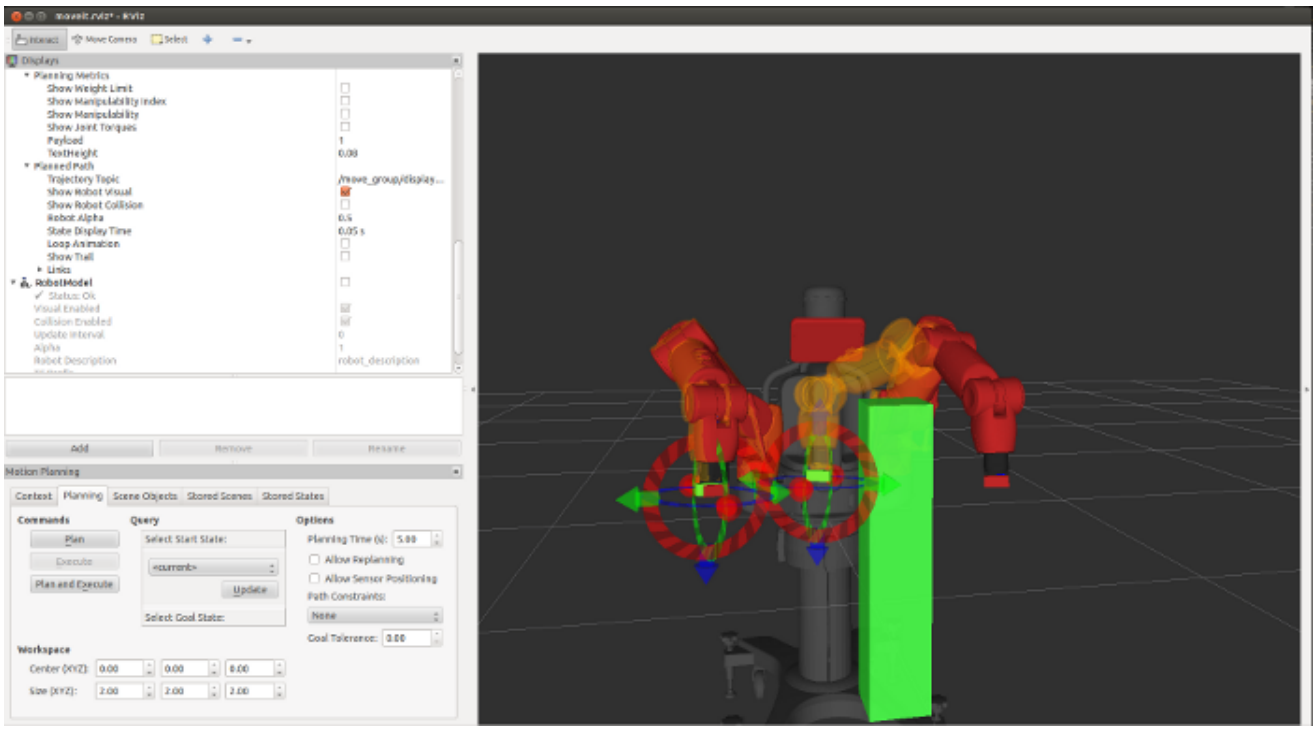


Figura V.11: Posição final com objeto Fonte: BAXTER2018

Selecionando *Planning* em *Motion Planning*, pode-se planejar uma nova trajetória que evitará a colisão com o objeto do ambiente (Figura V.12).

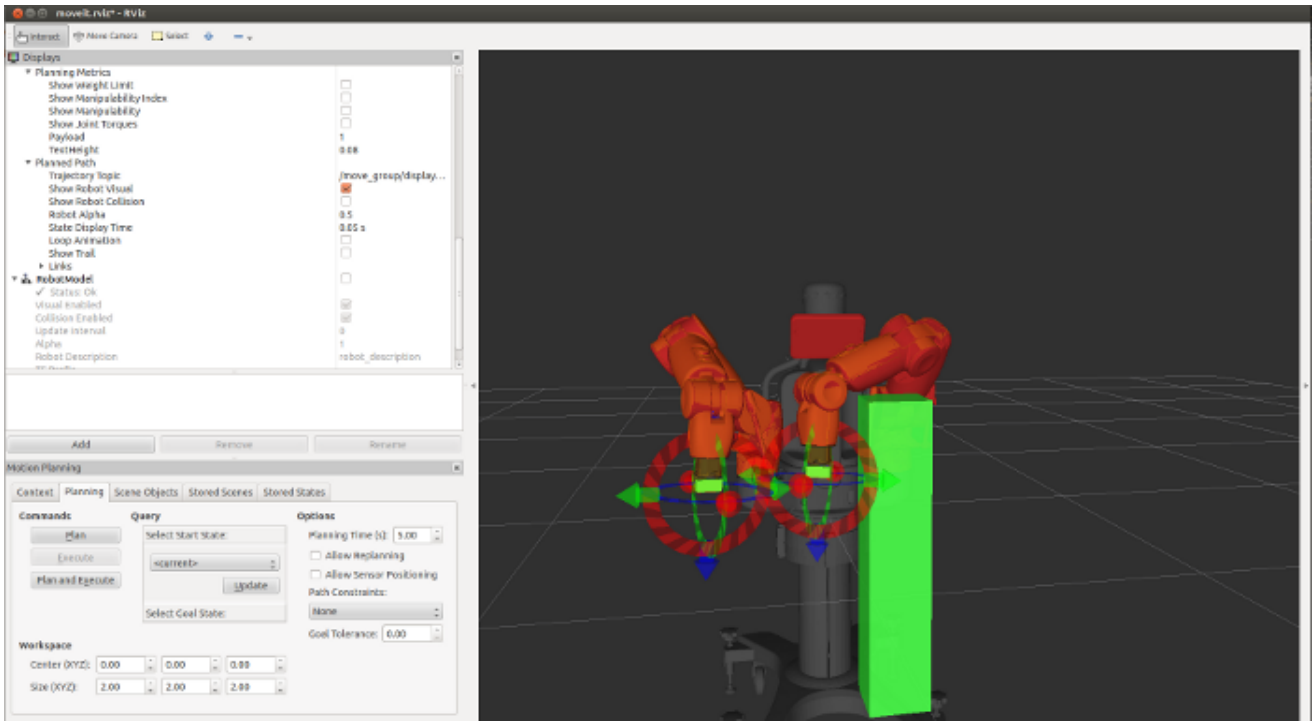


Figura V.12: Trajetória evitando o objeto Fonte: BAXTER2018

APÊNDICE VI – CÓDIGO FONTE

```
#!/usr/bin/env python
# coding: utf-8
"""
Baxter RSDK Inverse Kinematics Pick and Place
"""
#Importa a interface do baxter para acessar a classe da garra.

import argparse
import struct
import sys
import copy

import rospy
import rospkg

from gazebo_msgs.srv import (
    SpawnModel,
    DeleteModel,
)
from geometry_msgs.msg import (
    PoseStamped,
    Pose,
    Point,
    Quaternion,
)
from std_msgs.msg import (
    Header,
    Empty,
)
```

```
from baxter_core_msgs.srv import (
    SolvePositionIK,
    SolvePositionIKRequest,
)

import baxter_interface


#classes e funcoes


class PickAndPlace(object):
    # classe inicial
    def __init__(self, limb, hover_distance = 0.15, verbose=True):
        self._limb_name = limb # string
        self._hover_distance = hover_distance # em metros
        self._verbose = verbose # bool
        self._limb = baxter_interface.Limb(limb)
        self._gripper = baxter_interface.Gripper(limb)
        ns = "ExternalTools/" + limb + "/PositionKinematicsNode/IKService
        self._iksvc = rospy.ServiceProxy(ns, SolvePositionIK)
        rospy.wait_for_service(ns, 5.0)
        # verifica se o robo esta ativado
        print("Obtendo o estado do robo... ")
        self._rs = baxter_interface.RobotEnable(baxter_interface.CHECK_VE
        self._init_state = self._rs.state().enabled
        print("Ativando o robo... ")
        self._rs.enable()
    # classe posicao inicial do braco
    def move_to_start(self, start_angles=None):
        print("Moving the {0} arm to start pose...".format(self._limb_nam
```

```
        if not start_angles:
            start_angles = dict(zip(self._joint_names, [0]*7))
        self._guarded_move_to_joint_position(start_angles)
        self.gripper_open()
        rospy.sleep(1.0)
        print("Running. Ctrl-c to quit")

    def ik_request(self, pose):
        hdr = Header(stamp=rospy.Time.now(), frame_id='base')
        ikreq = SolvePositionIKRequest()
        ikreq.pose_stamp.append(PoseStamped(header=hdr, pose=pose))
        try:
            resp = self._iksvc(ikreq)
        except (rospy.ServiceException, rospy.ROSException), e:
            rospy.logerr("Service call failed: %s" % (e,))
            return False
        # Verifica se o resultado e valido

        resp_seeds = struct.unpack('<%dB' % len(resp.result_type), resp
        limb_joints = {}
        if (resp_seeds[0] != resp.RESULT_INVALID):
            seed_str = {
                        ikreq.SEED_USER: 'User Provided Seed',
                        ikreq.SEED_CURRENT: 'Current Joint Angles',
                        ikreq.SEED_NS_MAP: 'Nullspace Setpoints',
                       }.get(resp_seeds[0], 'None')
            if self._verbose:
                print("IK Solution SUCCESS - Valid Joint Solution Found
                      (seed_str)))

            limb_joints = dict(zip(resp.joints[0].name, resp.joints[0].
```

```
            if self._verbose:
                print("IK Joint Solution:\n{0}".format(limb_joints))
                print("------------------")
        else:
            rospy.logerr("INVALID POSE - No Valid Joint Solution Found.")
            return False
        return limb_joints

    def _guarded_move_to_joint_position(self, joint_angles):
        if joint_angles:
            self._limb.move_to_joint_positions(joint_angles)
        else:
            rospy.logerr("No Joint Angles provided for move_to_joint_posi

    def gripper_open(self):
        self._gripper.open()
        rospy.sleep(1.0)

    def gripper_close(self):
        self._gripper.close()
        rospy.sleep(1.0)

    def _approach(self, pose):
        approach = copy.deepcopy(pose)
        # deslocando acima da pose solicitada
        approach.position.z = approach.position.z + self._hover_distance
        joint_angles = self.ik_request(approach)
        self._guarded_move_to_joint_position(joint_angles)

    def _retract(self):
        # recupera a pose atual
```

```
        current_pose = self._limb.endpoint_pose()
        ik_pose = Pose()
        ik_pose.position.x = current_pose['position'].x
        ik_pose.position.y = current_pose['position'].y
        ik_pose.position.z = current_pose['position'].z + self._hover_d
        ik_pose.orientation.x = current_pose['orientation'].x
        ik_pose.orientation.y = current_pose['orientation'].y
        ik_pose.orientation.z = current_pose['orientation'].z
        ik_pose.orientation.w = current_pose['orientation'].w
        joint_angles = self.ik_request(ik_pose)
        # levanta
        self._guarded_move_to_joint_position(joint_angles)

    def _servo_to_pose(self, pose):
        # abaixa
        joint_angles = self.ik_request(pose)
        self._guarded_move_to_joint_position(joint_angles)

    def pick(self, pose):
        # abre a garra
        self.gripper_open()
        # movimenta sobre a posicao
        self._approach(pose)
        # movimenta na posicao
        self._servo_to_pose(pose)
        # fecha a garra
        self.gripper_close()
        # retira o objeto
        self._retract()

    def place(self, pose):
```

```
        # movimenta sobre a posicao
        self._approach(pose)
        # movimenta na posicaÃ£o
        self._servo_to_pose(pose)
        # abre a garra
        self.gripper_open()
        # retira a garra deixando o objeto
        self._retract()



def load_gazebo_models(table_pose=Pose(position=Point(x=1.0, y=0.0, z=0.0
                       table_reference_frame="world",
                       block2_pose=Pose(position=Point(x=0.7725, y=-0.1,
                       block2_reference_frame="world",
                       block3_pose=Pose(position=Point(x=0.7725, y=-0.1,
                       block3_reference_frame="world",
                       block4_pose=Pose(position=Point(x=0.7725, y=-0.1,
                       block4_reference_frame="world",
                       block_pose=Pose(position=Point(x=0.6725, y=0.1265,
                       block_reference_frame="world"):
    # Get Models' Path
    model_path = rospkg.RosPack().get_path('baxter_sim_examples')+"/model
    # Carrega Table SDF
    table_xml = ''
    with open (model_path + "cafe_table/model.sdf", "r") as table_file:
        table_xml=table_file.read().replace('\n', '')
    # Carrega Block2 URDF
    block2_xml = ''
    with open (model_path + "block/model2.urdf", "r") as block2_file:
        block2_xml=block2_file.read().replace('\n', '')
```

```
# Carrega Block3 URDF
block3_xml = ''
with open (model_path + "block/model3.urdf", "r") as block3_file:
    block3_xml=block3_file.read().replace('\n', '')

# Carrega Block4 URDF
block4_xml = ''
with open (model_path + "block/model4.urdf", "r") as block4_file:
    block4_xml=block4_file.read().replace('\n', '')

# Carrega Block URDF
block_xml = ''
with open (model_path + "block/model.urdf", "r") as block_file:
    block_xml=block_file.read().replace('\n', '')

# Spawn Table SDF
rospy.wait_for_service('/gazebo/spawn_sdf_model')
try:
    spawn_sdf = rospy.ServiceProxy('/gazebo/spawn_sdf_model', Spawn
    resp_sdf = spawn_sdf("cafe_table", table_xml, "/",
                         table_pose, table_reference_frame)
except rospy.ServiceException, e:
    rospy.logerr("Spawn␣SDF␣service␣call␣failed:␣{0}".format(e))
# Spawn Block2 URDF
rospy.wait_for_service('/gazebo/spawn_urdf_model')
try:
    spawn_urdf = rospy.ServiceProxy('/gazebo/spawn_urdf_model', Spa
    resp_urdf = spawn_urdf("block2", block2_xml, "/",
                           block2_pose, block2_reference_frame)
except rospy.ServiceException, e:
```

```
        rospy.logerr("Spawn URDF service call failed: {0}".format(e))
    # Spawn Block3 URDF
    rospy.wait_for_service('/gazebo/spawn_urdf_model')
    try:
        spawn_urdf = rospy.ServiceProxy('/gazebo/spawn_urdf_model', Spawn
        resp_urdf = spawn_urdf("block3", block3_xml, "/",
                               block3_pose, block3_reference_frame)
    except rospy.ServiceException, e:
        rospy.logerr("Spawn URDF service call failed: {0}".format(e))
    # Spawn Block4 URDF
    rospy.wait_for_service('/gazebo/spawn_urdf_model')
    try:
        spawn_urdf = rospy.ServiceProxy('/gazebo/spawn_urdf_model', Spawn
        resp_urdf = spawn_urdf("block4", block4_xml, "/",
                               block4_pose, block4_reference_frame)
    except rospy.ServiceException, e:
        rospy.logerr("Spawn URDF service call failed: {0}".format(e))
    # Spawn Block URDF
    rospy.wait_for_service('/gazebo/spawn_urdf_model')
    try:
        spawn_urdf = rospy.ServiceProxy('/gazebo/spawn_urdf_model', Spawn
        resp_urdf = spawn_urdf("block", block_xml, "/",
                               block_pose, block_reference_frame)
    except rospy.ServiceException, e:
        rospy.logerr("Spawn URDF service call failed: {0}".format(e))



def delete_gazebo_models():
```

```
    # This will be called on ROS Exit, deleting Gazebo models
    # Do not wait for the Gazebo Delete Model service, since
    # Gazebo should already be running. If the service is not
    # available since Gazebo has been killed, it is fine to error out
    try:
        delete_model = rospy.ServiceProxy('/gazebo/delete_model', Delete
        resp_delete = delete_model("cafe_table")
        resp_delete = delete_model("block")
        resp_delete = delete_model("block2")
        resp_delete = delete_model("block3")
        resp_delete = delete_model("block4")
    except rospy.ServiceException, e:
        rospy.loginfo("Delete Model service call failed: {0}".format(e)
def delete_gazebo_block2():

    try:
        delete_model = rospy.ServiceProxy('/gazebo/delete_model', Delete

        resp_delete = delete_model("block2")

    except rospy.ServiceException, e:
        rospy.loginfo("Delete Model service call failed: {0}".format(e)

def delete_gazebo_block3():

    try:
        delete_model = rospy.ServiceProxy('/gazebo/delete_model', Delete

        resp_delete = delete_model("block3")

    except rospy.ServiceException, e:
```

```
        rospy.loginfo("Delete␣Model␣service␣call␣failed:␣{0}".format(e))

def delete_gazebo_block4():

    try:
        delete_model = rospy.ServiceProxy('/gazebo/delete_model', DeleteM

        resp_delete = delete_model("block4")

    except rospy.ServiceException, e:
        rospy.loginfo("Delete␣Model␣service␣call␣failed:␣{0}".format(e))

def delete_gazebo_block():

    try:
        delete_model = rospy.ServiceProxy('/gazebo/delete_model', DeleteM

        resp_delete = delete_model("block")

    except rospy.ServiceException, e:
        rospy.loginfo("Delete␣Model␣service␣call␣failed:␣{0}".format(e))



def main():
    """RSDK Inverse Kinematics Pick and Place Example

    Um exemplo de Pick and Place usando o Rethink Inverse Kinematics Serv
    alcancar o objetivo

    """
    rospy.init_node("ik_pick_and_place_demo")
```

```
# Carregar modelos do gazebo
load_gazebo_models()
# Remover modelos da cena no desligamento
rospy.on_shutdown(delete_gazebo_models)

# Aguardando a inicializaÃ§Ã£o do emulador
rospy.wait_for_message("/robot/sim/started", Empty)

limb = 'left'
hover_distance = 0.15 # meters
# Angulos de articulacao iniciais para o braco esquerdo
starting_joint_angles = {'left_w0': 0.6699952259595108,
                         'left_w1': 1.030009435085784,
                         'left_w2': -0.4999997247485215,
                         'left_e0': -1.189968899785275,
                         'left_e1': 1.9400238130755056,
                         'left_s0': -0.08000397926829805,
                         'left_s1': -0.9999781166910306}
pnp = PickAndPlace(limb, hover_distance)
# Uma orientacao para os dedos da garra ficarem acima e paralelos a
overhead_orientation = Quaternion(
                           x=-0.0249590815779,
                           y=0.999649402929,
                           z=0.00737916180073,
                           w=0.00486450832011)
block_poses = list()
# A pose dos blocos na localizaÃ§Ã£o inicial.

block_poses.append(Pose(
    position=Point(x=0.67, y=0.15, z=-0.135),
    orientation=overhead_orientation))
```

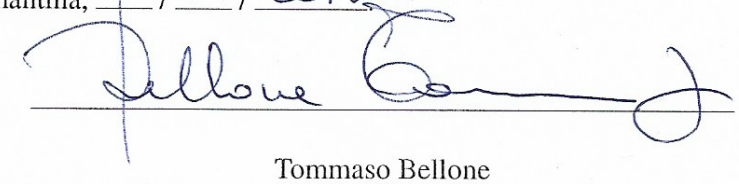
```
block_poses.append(Pose(
    position=Point(x=0.75, y=0.0, z=-0.135),
    orientation=overhead_orientation))

block_poses.append(Pose(
    position=Point(x=0.75, y=0.1, z=-0.135),
    orientation=overhead_orientation))
# Movendo para os angulos de partida desejados
pnp.move_to_start(starting_joint_angles)

idx = 0
while not rospy.is_shutdown():

    print("\nPicking...")
    pnp.pick(block_poses[idx])
    print("\nPlacing...")
    idx = (idx+1) % len(block_poses)

    pnp.place(block_poses[idx])
    if (idx)==1:
        delete_gazebo_block2()
    elif (idx)==2:
        delete_gazebo_block3()



    if (idx)==0:
        delete_gazebo_block4()
        rospy.sleep(0.5)
        delete_gazebo_block()
        rospy.sleep(0.5)
```

```
            load_gazebo_models()
            int(input("Digite um numero para continuar"))


    return 0



if __name__ == '__main__':
    sys.exit(main())
```

AUTORIZAÇÃO

Autorizo a reprodução e/ou divulgação total ou parcial do presente trabalho, por qualquer meio convencional ou eletrônico, desde que citada a fonte.

Diamantina, 21 / 01 / 2019



Tommaso Bellone

bellonetom@gmail.com

Universidade Federal dos Vales do Jequitinhonha e Mucuri

Campus Alto do Jacuba - CEP: 39100-000 - Diamantina - MG - Brazil