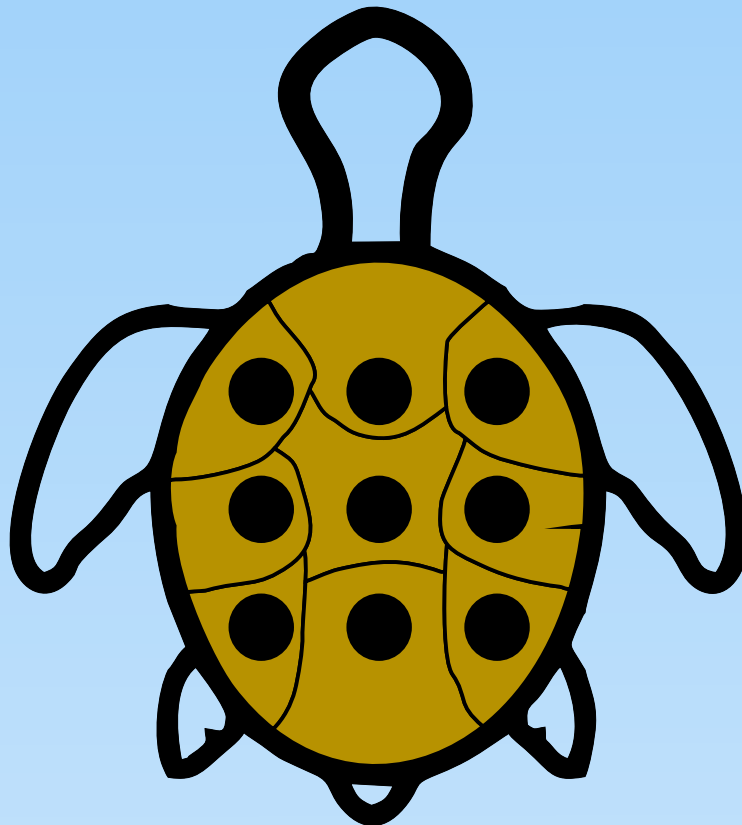




The Turtles

◊ guia prático e introdutório de
simulações em robótica com ROS



2021

Arthur H. D. Nunes

Prefácio

Experimentos são imprescindíveis para validar metodologias propostas na área da robótica. Diante disso, esta apostila trará um compilado e síntese de tutoriais sobre ferramentas utilizadas em nossos laboratórios para simular, visualizar e tratar experimentos robóticos.

Desta forma, servirá como guia prático e introdutório para novos integrantes do laboratório e também para aqueles que desejem aprender sobre simulações robóticas.

É recomendado que o leitor já saiba programar, ou pelo menos tenha noções de Python, C/C++ e MATLAB, uma vez que serão amplamente utilizados no decorrer da apostila.

Ademais, para as próprias simulações e experimentos, é desejável conhecimento de planejamento de movimento e controle de robôs, mas essa demanda não interferirá no entendimento do funcionamento das ferramentas, apenas nos usos e aplicações.

Dentre as principais ferramentas utilizadas, destacam-se o Robot Operating System (ROS), o CoppeliaSim e o Matrix Laboratory (MATLAB).

Para facilitar a portabilidade dos arquivos usados nessa apostila, eles podem ser encontrados em github.com/ArthurHDN/ApostilaTheTurtles.

Esta apostila é dedicada a equipe AAI Robotics, que conquistou o terceiro lugar na competição Rosi Challenge, dentro do Simpósio Brasileiro de Automação Inteligente (SBAI) 2019 (<https://www.sbai2019.com.br/rosi-challenge>).

É com muito esmero e dedicação que realizamos esta atividade.

2021,
Arthur H. D. Nunes,
Adriano M. C. Rezende,
Álvaro Rodrigues Araújo
Orientador: Luciano C. A. Pimenta

LABORATÓRIO DE SISTEMAS DE COMPUTAÇÃO E ROBÓTICA
[HTTP://CORO.CPDEE.UFMG.BR/](http://coro.cpdee.ufmg.br/)
LABORATÓRIO DE MECATRÔNICA, CONTROLE E ROBÓTICA
[HTTP://MACRO.PPGEE.UFMG.BR/](http://macro.ppgEE.ufmg.br/)
PROGRAMA DE EDUCAÇÃO TUTORIAL DA ENGENHARIA ELÉTRICA
[HTTP://WWW.PETEE.CPDEE.UFMG.BR/](http://www.petEE.cpdee.ufmg.br/)

UNIVERSIDADE FEDERAL DE MINAS GERAIS



Sumário

I	Preparação	
1	Linux	9
1.1	O Sistema Operacional	9
1.2	Instalação	10
1.2.1	Máquina Virtual	10
1.2.2	Windows Subsystem Linux	10
1.2.3	Dual Boot	11
1.3	Terminal	13
1.3.1	Atalhos	14
1.3.2	Navegação Em Pastas	14
1.3.3	Opções Com Arquivos	15
1.3.4	Permissões	15
1.3.5	Programas em Segundo Plano	16
1.3.6	Instalação	17
1.3.7	Resumo	17
2	Introdução	19
2.1	Sobre o ROS	19
2.2	Instalação	20
2.3	Área de Trabalho e Pacotes	22
2.4	Navegação	24

II**Conceitos Básicos**

3	Nós e Tópicos	27
4	Criando nós	35
4.1	Publicar	35
4.2	Subscrever	39
4.3	Compilação e Execução	42
5	Hello World Turtlesim	45
5.1	Objetivo Inicial	45
5.2	Refinando o método	48
5.3	Alvo Variante	50
6	Ferramentas	53
6.1	Serviços e Mensagens	53
6.1.1	Entendendo Arquivos .msg e .srv	53
6.1.2	Preparação	54
6.1.3	Utilização de .srv	56
6.2	Parâmetros	60
6.3	Servidor de Reconfiguração Dinâmica	60
6.3.1	Arquivo .cfg	61
6.3.2	Configurando o Nó Servidor Dinâmico	62
6.3.3	Exemplo de Cliente	63
6.3.4	Exemplo de Cliente e Serviço de Tipo Trigger	64
6.4	Remap e Namespace	67
6.5	Launch	68
6.6	Múltiplas Máquinas	70

III**Simulações**

7	Stage	75
8	RViz	83
9	Gazebo	97
10	CoppeliaSim	107
10.1	Introdução	107
10.2	Instalação	108
10.3	Interface Gráfica do Usuário	108
10.4	Comunicação Remota	109
10.4.1	ROS	109
10.4.2	MATLAB	111

11	Teleop ROSI	113
11.1	Baixo Nível	113
11.2	Launch	116
12	Cenário de um Manipulador	119
12.1	Cenário	120
12.2	Controle Externo	124
13	Controle de Quadrotor	127
13.1	Teoria	127
13.2	Cenário	129
13.3	Implementação	131

IV

Índice

Referências Bibliográficas	141
Repositórios	143
Apêndice I - AAI Robotics	145
Apêndice II - CORO	147
Instalação	147
Joystick	148
iRobot Create	149
Câmeras	150



Preparação

1	Linux	9
1.1	O Sistema Operacional	
1.2	Instalação	
1.3	Terminal	
2	Introdução	19
2.1	Sobre o ROS	
2.2	Instalação	
2.3	Área de Trabalho e Pacotes	
2.4	Navegação	



1. Linux

Antes de entender e usar o ROS, deve-se preparar e familiarizar-se com o ambiente Linux, crucial para seu funcionamento. Este capítulo, portanto, trará tutoriais e noções de Linux, mais especificamente, da versão Ubuntu 16.04.

Há maneiras de se colocar o ROS diretamente no sistema operacional Windows, mas ainda não são muito difundidas e o uso do Linux é estritamente recomendado.

Desta forma, mesmo que o usuário já tenha o Linux e o ROS instalado, este capítulo pode trazer alguns conceitos úteis, como entender alguns comandos.

1.1 O Sistema Operacional

Sistemas Operacionais são softwares que realizam intercomunicação entre usuário e hardware. Alguns deles são: Windows, MacOS e Linux. O Linux é gratuito, isto é, qualquer um pode baixar o núcleo, também conhecido como kernel, estudar, modificar e distribuir livremente, de acordo com os termos da licença General Public License (GPL). O kernel foi desenvolvido pelo programador finlandês Linus Torvalds.

A distribuição foco da apostila é a Ubuntu. É uma distribuição bem conceituada, popular e robusta. A palavra *ubuntu* é da língua Zulu, de origem africana, Não possui tradução direta, mas em tradução livre pode significar generosidade, compaixão ou solidariedade. Sua ideia também pode ser expressa por "sou o que sou pelo que nós somos".



Figura 1.1.1: *Ubuntu 16.04 LTS*

A versão **Ubuntu 16.04 LTS**, como mostra a Fig. 1.1.1, será utilizada porque é, atualmente, a que possui melhor compatibilidade com o software **ROS Kinetic Kame** Fig. 2.1.2.

1.2 Instalação

Esta seção é apenas para quem ainda não tem instalado o Linux, pois abordará alguns métodos de instalação do Ubuntu 16.

1.2.1 Máquina Virtual

Uma maneira de utilizar o Ubuntu dispondo de outros sistemas operacionais é a criação de uma máquina virtual. Dessa forma, é criada uma simulação de uma máquina dentro de outra máquina.

Recomenda-se este artifício apenas para testar ou se acostumar com o Linux. Para um melhor desempenho e uma melhor experiência com o ROS, é melhor usar o Ubuntu diretamente. Uma boa forma sem apagar o Sistema Operacional atual é realizando Dual Boot, explicado na próxima subseção.

Para a criação é necessário instalar algum software de máquina virtual, VirtualBox e VMware são populares.

1.2.2 Windows Subsystem Linux

Para usar o WSL é necessário ter Windows 10 na máquina, porque somente esse tem linux em sua base. O método consiste em baixar o ubuntu diretamente no Windows, e usar a extensão WSL do *Visual Studio Code*. Esse é recomendado para usuários que usam muitos recursos do Windows e alguns do Linux, como por exemplo, comandos gits podem ser usados após a instalação desse método.

Em questões de desempenho, é melhor que a máquina virtual, pois os recursos do Windows são disponibilizados diretamente para o Linux, mas ainda não tão bom quanto dual boot, porque há também um Windows sendo executado. Ademais, nesse é necessário configurar um XDisplay para ter acesso a qualquer interface gráfica.

O tutorial será dividido em quatro partes. Sugestão: visitar também o site <https://code.visualstudio>.

com/docs/remote/wsl

Parte I: Instalação do WSL

<https://docs.microsoft.com/pt-br/windows/wsl/install-win10>

- 1 Abra o PowerShell como administrador e insira o comando:

```
dism.exe /online /enable-feature /featurename:  
Microsoft-Windows-Subsystem-Linux /all /norestart
```

- 2 Abra a Microsoft Store e instale a distribuição do Linux desejada.
- 3 Execute a distribuição instalada e crie uma nova conta inserindo nome de usuário e senha.

Parte II: Instalação do VS Code

<https://code.visualstudio.com/>

- 1 Instale o Visual Studio Code no Windows.

Parte III: Instalação da extensão Remote Developent para o VS Code

<https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensi>

- 1 Procure a extensão e faça o download no VS Code.
- 2 Para utilizar a extensão, clique no símbolo de >< no canto inferior esquerdo e depois clique em **Remote-WSL: New Window**.

Parte IV: Configurando um XDisplay

- 1 Faça a instalação do Xming, XSrv ou similar <https://sourceforge.net/projects/xming/>
- 2 Execute o XLaunch ou similar.
- 3 Configure para **Multiple Windows** e **Display Number = 0**. OBS: Você pode salvar essas configurações e permitir com que o XLaunch inicie sempre com o Windows ou pode optar por executá-lo manualmente sempre que for utilizar o recurso.
- 4 Execute o Remote-WSL, abra um novo terminal e insira o comando

```
code ~/.bashrc
```

- 5 Adicione ao final do arquivo a seguinte instrução:

```
export DISPLAY=localhost:0
```

1.2.3 Dual Boot

O Dual Boot é uma maneira melhor de usar o Ubuntu. O método não deve apagar os arquivos já existentes no HD, mas por precaução é recomendado realizar o backup de arquivos importantes em HD externo ou Nuvem.

Tutorial de instalação do Ubuntu em dual boot com Windows 10:

Sugestão: visitar também o site <https://www.tecmint.com/install-ubuntu-alongside-with-windows-dual->

- Antes da instalação, deve desativar a inicialização rápida do Windows, o que poderá dificultar a fácil troca de sistema operacional. Para isso vá em "Painel de Controle", "Opções de Energia", "Escolher a função dos botões de energia", "Alterar configurações não disponíveis no momento", desmarque a opção "Ligar inicialização rápida (recomendado)" e clique em "Salvar alterações". Agora poderá prosseguir com a instalação:
- 1 É necessário baixar o arquivo ISO do Ubuntu 16.04 LTS, que está disponível gratuitamente em <https://www.ubuntu.com/download/desktop>
 - 2 Crie um pendrive bootável com o Ubuntu.
 - 3 Abra o gerenciador de disco. Para isso aperte "Windows" + "R", digite "diskmgmt.msc" e pressione "Enter" ou aperte com o botão direito em Este Computador, Gerenciar e selecionar Gerenciador de Disco

- 4 Clique com o botão direito na unidade que deseja particionar e selecione "Diminuir Volume". Recomendamos uma partição de 100Gb = 102400Mb. Após, deve aparecer um espaço de 100Gb - ou do tamanho liberado pelo usuário - não alocado. É onde será instalado o Ubuntu.
- 5 Desligue a máquina, plugue o pendrive e ligue a máquina. Abra a BIOS de seu computador. Cada marca possui um atalho diferente, mas geralmente é pressionar algumas das teclas "Delete", "F12", "F10", "Esc", enquanto o computador inicia. Este passo pode ser um pouco complicado dependendo do fabricante do computador. O objetivo é entrar na BIOS e fazer uma seleção de boot para escolher o pendrive bootável. Para isso, ao entrar no menu, procure por algo similar a "boot setup", "boot option" ou pela opção de executar o pendrive.
- 6 Selecione a língua que deseja e em seguida, em Instalar Ubuntu.
- 7 Selecione cuidadosamente a partição que irá receber o Ubuntu e clique em "+" para adicionar nova partição.
- 8 No tamanho, recomendamos que seja igual ou o dobro da quantidade de memória RAM de sua máquina. Tipo: "Lógica"; Localização para nova partição: "Início deste espaço"; Usar como: "Área de troca (swap)".
- 9 Novamente, selecione o espaço que sobrou e clique em "+" para adicionar nova partição.
- 10 Tamanho deixe o resto que sobrou. Tipo: "Lógica"; Localização para nova partição: "Início deste espaço"; Usar como: "Sistema de arquivos com "journaling"; Ponto de montagem: "/".
- 11 Selecione a última partição realizada e prossiga com a instalação.

Feito isso, sempre que a máquina for reiniciada deve aparecer a opção de escolha do sistema operacional pelo Grub do Ubuntu.

Talvez, a hora do Windows esteja sempre três horas adiantada. É fácil de resolver:

- Inicie o Windows.
 - 1 Pressione "Windows" + "R", digite "regedit" e pressione "Enter" para abrir o editor de registro.
 - 2 Siga até "HKEY_LOCAL_MACHINE\ SYSTEM\ ControlSet001\ Control\ Time-ZoneInformation".
 - 3 Clique com o botão direito em "TimeZoneInformation", clique em "Novo" e em "Valor DWORD (32 bits)". Dê o nome de "RealTimeIsUniversal".
 - 4 Abra o "RealTimeIsUniversal", altere seu valor de 0 para 1 e clique em "OK".
 - 5 Confirme as alterações, feche e reinicie o sistema.

Adicionalmente, pode ser desejável deixar a inicialização do Windows como padrão. Para isso, deve-se mudar a ordem ou a seleção padrão da opção no grub do Ubuntu. Também é um procedimento fácil:

- Inicie o Ubuntu.

- 1 Abra o terminal e digite

```
$ sudo gedit /etc/default/grub
```

- 2 Com o arquivo aberto, localiza a linha onde está escrito:

```
GRUB_DEFAULT=0
```

Esta linha marca a opção que será selecionada ao iniciar a máquina. Como está marcado 0, a primeira opção, que estará o Ubuntu, será selecionada por padrão.

- 3 O usuário deverá alterar o valor 0 para o valor desejado, lembrando que a enumeração começa do 0, ou seja, caso seja desejado, por exemplo, a terceira opção, o usuário deverá colocar o valor 2.

- 4 Salve as alterações feitas no arquivo e feche-o.

- 5 Digite o comando no terminal:

```
sudo update-grub
```

Outro tutorial de como fazê-lo também está disponível em <http://tipsonubuntu.com/2016/07/20/grub2-boot-order-ubuntu-16-04/>

1.3 Terminal

Os terminais Linux serão bastante utilizados. Para isso é bom ter uma noção de comandos e de como utilizá-lo. O Ubuntu vem, por padrão, com o Gnome Terminal instalado.

DICA: Outros terminais também podem ser instalados de acordo com a preferência do usuário. Terminator é uma boa opção, porque permite várias abas e até mesmo dividir uma aba em várias.

```
$ sudo apt-get install terminator
```

Os terminais executam códigos conhecidos como *Bash* ou *Shell*, que manipulam dados do usuário a partir de linhas de comando. Toda vez que um novo terminal é aberto, ele executa o arquivo oculto `~/.bashrc`.

Para melhor entendimento dos exemplos de códigos da apostila, quando houver a marcação "\$" o comando deverá ser executado no terminal. O que estiver entre os símbolos menor e maior que «>» deve ser substituído na hora da execução. Exemplo, ao ser indicado:

```
$ mkdir <nome>
```

Deve ser digitado no terminal algo como:
mkdir minhapasta

No fim da seção, há a tabela 1.3.1 com os principais atalhos e comandos.

1.3.1 Atalhos

Alguns atalhos são práticos e ajudam a ter uma maior fluência ao usar o Terminal. Fique atento pois Ctrl + C não copia, e sim interrompe a execução corrente. Portanto, para copiar e colar deve-se pressionar Ctrl + Shift + C e Ctrl + Shift + V, respectivamente. Os atalhos Ctrl + Shift + N e Ctrl + Shift + T abrem, em ordem, uma nova janela e nova aba. Também é possível realizar essas ações clicando em "Arquivo" no canto superior direito da janela. Ainda nesse canto, em "Editar" depois "Preferências" é possível configurar cores, fontes, estilos e outros da forma que o usuário preferir. Ctrl + "+" aumenta a letra, enquanto Ctrl + "-" a diminui, podendo também ser feito clicando em "Ver".

Um atalho muito útil é o *tab*, que completa automaticamente o comando ou sugere as opções de completar quando existe mais de uma opção.

1.3.2 Navegação Em Pastas

Saber navegar por arquivos utilizando interfaces de comando de linha, ou *command line interface* (CLI) ao invés de interfaces gráficas ou *graphical user interface* (GUI) é essencial. Tanto que uma característica dos usuários de Linux é o maior uso de CLIs quando comparado aos usuários de Windows.

Para saber o caminho para o diretório corrente deve-se usar

```
$ pwd
```

de *print working directory*.

O comando

```
$ cd <dir>
```

de *change directory*, navega para o diretório *dir*. Importante lembrar que em todas as pastas existem o arquivo "." e o arquivo "..", dos quais o primeiro contém o endereço para o diretório corrente e o segundo o endereço para o diretório onde se encontra o diretório corrente. Logo o comando

```
$ cd ..
```

volta um diretório. O comando

```
$ cd -
```

volta para o diretório imediatamente anterior ao que estava.

Para se criar uma pasta pode se usar o comando

```
$ mkdir <dir>
```

de *make directory*.

Para listar os arquivos e diretórios dentro do diretório corrente, deve se usar

```
$ ls
```

de *list*. Frequentemente, comandos que usamos necessitam de parâmetros ou podem ser modificados por eles. Um parâmetro é indicado por um ou dois traços simples, sendo assim, para listarmos os arquivos e diretórios e vermos mais detalhes, devemos usar

```
$ ls -l
```

1.3.3 Opções Com Arquivos

Para que o terminal imprima mensagens, pode ser utilizado o comando

```
$ echo <mensagem>
```

Ele também serve para redirecionar a mensagem para um arquivo, desta vez, sem que o terminal a exiba

```
$ echo <mensagem> > <arquivo>
```

O terminal também pode ser usado para abrir aplicativos, como o *gedit*, um editor de texto que já vem no Ubuntu. Podemos fazer então

```
$ gedit <arquivo>
```

e assim, abriremos o arquivo no *gedit*. O arquivo *.bashrc* é o roteiro, ou *script* executado sempre que o terminal é aberto. Quando estivermos na parte de ROS, podemos editá-lo para já deixar nossa *workspace* sourceada e navegar até ela. Um exemplo desse uso é escrever no final de *.bashrc*, usando o *gedit*, *echo Ola Mundo*, assim, sempre que um novo terminal for aberto, Ola Mundo será exibido.

O comando

```
$ cat <arquivo>
```

pode ser utilizado para visualizar o conteúdo de um arquivo no terminal.

Para copiarmos ou movermos arquivos e pastas, usamos os comandos

```
$ cp <origem> <destino>
```

```
$ mv <origem> <destino>
```

respectivamente, de *copy* e *move*. Caso o comando *mv* seja usado para mover um arquivo para o mesmo diretório, essa ação pode ser interpretada como uma renomeação do arquivo, uma vez que todo o conteúdo será transferido para outro arquivo com nome diferente - caso o usuário queira - e na mesma localização.

Para removermos arquivos, utilizamos o comando

```
$ rm <arquivo>
```

Podemos utilizá-lo também para remover diretórios, usando um parâmetro para forçar a remoção, *-rf*.

```
$ rm -rf <dir>
```

1.3.4 Permissões

Por padrão, os arquivos criados por um usuário pertencem ao mesmo, e por segurança, as permissões desses arquivos podem ser alteradas. O usuário *root* é o aquele com acesso e permissão a todo sistema. Usando o comando *ls -l* visto anteriormente, os arquivos são listados com detalhamento. Os primeiros caracteres dos detalhes de um arquivo devem parecer com:

```
drwxr-xr-x 2 user user
```

O primeiro *user* é o usuário dono do arquivo, enquanto que o segundo é o grupo de usuários a que pertence. Por padrão, quando não se criam grupos, cada usuário terá seu grupo com seu respectivo nome.

Já os dez primeiros caracteres representam as permissões. O primeiro informa se é um arquivo - ou se é um diretório *d*. Os nove subsequentes são três repetições de *rwX*, sendo alguns substituídos por traço simples, indicando que não possui a permissão da ação que seria a letra correspondente (*r*, *w* ou *X*). Portanto, a primeira tripla são as permissões do dono, a segunda as permissões do grupo e a terceira as permissões de todos. A letra *r* indica permissão para leitura, *w* para escrita e *X* para execução.

Para alterarmos as permissões de um arquivo, podemos utilizar o comando `chmod <valor1><valor2><valor3> <arquivo>`, em que cada valor é a soma das permissões concebidas ao usuário, grupo e todos, respectivamente. As permissões tem valor *r* = 4, *w* = 2 e *X* = 1. Logo, para concebermos permissão total ao dono, grupo e todos, usamos

```
$ chmod 777 <arquivo>
```

Comando que será usado quando abordarmos ROS. Eventualmente, quando formos executar programas feitos em Python, precisaremos conceber permissão de execução ao arquivo. Podemos fazê-lo com

```
$ chmod +X <arquivo>
```

O parâmetro `-R` pode ser usado para que a alterações de permissões afetem todas as pastas e arquivos dentro da pasta especificada.

Para executar comandos dos quais não se tem permissão, pode ser usado o comando `sudo`, de *super user do*, antes do comando desejado.

```
$ sudo comando
```

O sistema irá pedir a senha para verificação.

Para alterar para o usuário *root*, pode-se usar o comando

```
$ sudo su
```

1.3.5 Programas em Segundo Plano

Ao executarmos um programa pelo terminal, como `gedit` ou navegador Firefox, percebemos que aquela aba será ocupada com o programa, não permitindo outros comandos. Para executarmos um programa em segundo plano basta utilizarmos `&` após o nome do programa.

```
$ <programa> &
```

Note que um número ID será mostrado em seguida, este ID pode ser usado para terminar a execução do programa por meio do comando

```
$ kill <ID>
```

Para visualizar os atuais processos e algumas informações do sistema, pode-se utilizar o comando `top`.

```
$ top
```

DICA: Outra forma similar de visualizar os processos é usando o `htop`. Talvez seja necessário instalá-lo via "sudo apt-get install htop".

1.3.6 Instalação

O comando para instalação e remoção de arquivos é *apt-get*. Ele sempre precisará de permissão *root*, logo é precedido de *sudo*. Sendo assim, é escrito da seguinte forma:

```
$ sudo apt-get <update/install/remove> <programa>
```

A ação pode ser *update* - nesse caso não é necessário especificar o programa. Haverá a atualização dos programas instalados. Pode ser também *install* para instalar. E pode ser *purge* ou *remove* para remover.

1.3.7 Resumo

A tabela 1.3.1 apresenta os atalhos e comandos resumidos.

Tabela 1.3.1: *Atalhos e Comandos*

Tipo	Atalho/Comando	Descrição
Atalho	Ctrl + Shift + N	Nova janela
Atalho	Ctrl + Shift + T	Nova aba
Atalho	Ctrl + C	Matar a execução
Atalho	Ctrl + Shift + C	Copiar
Atalho	Ctrl + Shift + V	Colar
Atalho	Ctrl + "+"	Aumenta a letra
Atalho	Ctrl + "-"	Diminui a letra
Atalho	Tab	Completa o comando
Atalho	Tab Tab	Sugere comandos
Comando	clear	Limpa a tela
Comando	sudo <comando>	Executar com permissão de root
Comando	pwd	Caminho para o diretório corrente
Comando	cd <dir>	Navega até dir
Comando	cd ..	Volta um diretório
Comando	cd -	Volta até o local anterior
Comando	ls	Lista arquivos e diretórios
Comando	echo	Exibe mensagens
Comando	cp <origem> <destino>	Copia da origem para o destino
Comando	mv <origem> <destino>	Move da origem para o destino
Comando	chmod <valores> <arquivo>	Modifica as permissões do arquivo
Comando	<programa> &	Executa em segundo plano
Comando	kill <ID>	Termina o processo
Comando	top	Visualiza processos e informações
Comando	sudo apt-get update	Atualiza os programas
Comando	sudo apt-get <ação> <programa>	Instala ou remove o programa



2. Introdução

Após a instalação e familiarização com o Linux, pode-se proceder para a preparação do ROS. Desta forma, este capítulo cobrirá a introdução e preparação do ambiente ROS. O entendimento de seu funcionamento e como utilizá-lo será melhor explicado pelo próximo.

2.1 Sobre o ROS

O *Robot Operating System* (ROS) é uma *framework* para desenvolvimento de softwares robóticos que visa facilitar a criação de complexos e robustos comportamentos de máquinas, provido de bibliotecas, ferramentas, convenções, atalhos, simuladores e tutoriais. Ele permite a execução de vários programas e estabelece comunicações entre eles. O seu núcleo, o *roscore*, é o primeiro programa que deve ser executado. Assim em diante, os programas iniciados conectam-se a ele e registram detalhes sobre os tipos de dados que desejam enviar ou receber. Como ilustra a Fig. 2.1.1. A estrutura final pode ser representada por um grafo. Nesta apostila, a representação do *roscore* será omitida para simplificação do grafo. O programa *talker* indica ao *roscore* que irá enviar um tipo de mensagem, enquanto o programa *listener* indica que irá receber este mesmo tipo. Assim, o *roscore* estabelece uma comunicação entre eles.

Ao iniciar o programa, ele procura uma variável de ambiente chamada *ROS_MASTER_URI*, que contém uma string do tipo `http://hostname:11311/`, implicando que há um *roscore* sendo executado na porta 11311 em algum *host* disponível na rede.

CURIOSIDADE: A porta 11311 é escolhida como default porque, na época do desenvolvimento do ROS, era um palíndromo primo inutilizado. Qualquer porta pode ser usada (1025 até 65535). Portas diferentes podem ser especificadas, permitindo que sistemas ROS coexistam em uma mesma rede.

À um programa é dado o nome de **nó**, e à uma comunicação de **tópico**.

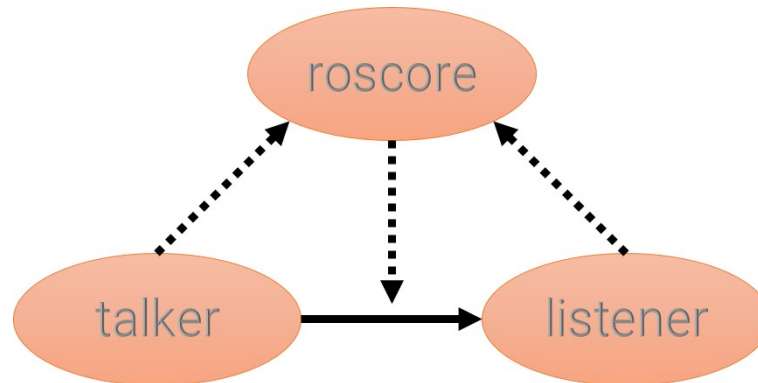


Figura 2.1.1: Grafo de um Listener e um Talker

Nesta apostila, será utilizada a versão Kinetic Kame do ROS, Fig. 2.1.2, melhor compatível com o sistema operacional Ubuntu 16.04 LTS.



Figura 2.1.2: ROS Kinetic Kame

CURIOSIDADE : Você pode acompanhar as distribuições do ROS no link <http://wiki.ros.org/Distributions>. Cada distribuição tem uma tema e uma tartaruga ícone.

2.2 Instalação

Para a instalação do ROS de forma recomendada, deve-se apenas usar no terminal os comandos listados a seguir. O comando `sudo apt-get install ros-kinetic-desktop-full` pode demorar um pouco. Os comandos são explicados em sequência e recomenda-se ler a explicação antes de usá-los. O guia do ROS para esta instalação está disponível em <http://wiki.ros.org/kinetic/Installation/Ubuntu>

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

$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com
:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

$ sudo apt-get update

# Pode demorar um pouco
$ sudo apt-get install ros-kinetic-desktop-full

$ sudo rosdep init

$ rosdep update

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

$ source ~/.bashrc

$ sudo apt install python-rosinstall python-rosinstall-
generator python-wstool build-essential
```

Antes da instalação, propriamente dita, deve-se fazer com que o computador permita a instalação do software de packages.ros.org, que é o primeiro comando. Ele configura suas chaves de instalação. Caso dê erro no segundo comando, verifique se a chave usada é a mesma que consta no link oficial de instalação, indicado acima.

Em sequência, deve-se verificar se os pacotes estão atualizados, pelo terceiro comando. Já o quarto pode demorar um pouco.

DICA: Nele é recomendado instalar o *ros-desktop-full*, mas pode ser substituído por instalar apenas ROS, *rqt*, *rviz*, bibliotecas genéricas ou pacotes ROS, *build* e bibliotecas de comunicação, sem ferramentas GUI, por meio de um dos comandos:

```
$ sudo apt-get install ros-kinetic-desktop

$ sudo apt-get install ros-kinetic-ros-base
```

Desta forma, pacotes adicionais podem ser encontrados e instalados com os seguintes comandos:

```
$ apt-cache search ros-kinetic

$ sudo apt-get install ros-kinetic-<pacote>
```

Em seguida, deve-se iniciar o *rosdep*, pelos dois próximos comandos. Ele permite fáceis instalações de pacotes e dependências ROS e é requerido por alguns componentes do núcleo ROS.

O penúltimo comando da *source* no *.bashrc* que foi modificado pelo anterior, por sua vez adiciona outro *source*, da biblioteca ROS ao arquivo, para que seja automático sempre que um novo terminal for aberto. Por fim, o último instala algumas bibliotecas de Python necessárias.

DICA: Para alguns tutoriais será necessário instalar o pacote *ros-tutorials*:

```
$ sudo apt-get install ros-kinetic-ros-tutorials
```

2.3 Área de Trabalho e Pacotes

Para começar a usar o ROS é necessário criar áreas de trabalho ou **workspaces** e pacotes ou **packages**. Um bom tipo de workspace para começar é o tipo **catkin_make**. Posteriormente pode-se usar uma do tipo **catkin build**.

Para criar uma workspace é necessário criar o diretório e compilá-lo. Como utilizaremos *catkin_make*, o comando para compilação é também *catkin_make*. O nome pode ser escolhido de acordo com o gosto do usuário. Nomearemos a nossa com o nome padrão, **catkin_ws**.

```
$ mkdir -p ~/catkin_ws/src
```

```
$ cd ~/catkin_ws/src
```

```
$ catkin_init_workspace
```

Ao utilizar a workspace, isto é, seus nós e *launchs*, deve-se usar o *source* sempre que abrir um novo terminal.

```
$ source devel/setup.bash
```

O comando pode, eventualmente, ser esquecido. Por isso, caso tenha uma única workspace, ela pode ser automaticamente sourceada colocando o código no arquivo *.bashrc*, assim como foi feito anteriormente. Para isso, pode-se utilizar o comando a seguir, ou adicionar manualmente ao arquivo.

```
$ echo "source devel/setup.bash" >> ~/.bashrc
```

Os pacotes são pastas nas quais se encontram os arquivos executados pelo ROS. Eles ficam dentro da pasta *src* da workspace e devem conter pelo menos dois arquivos para se caracterizarem como tal: *CMakeLists.txt* e *package.xml*. A estrutura de pacotes de uma workspace é ilustrada pela Fig. 2.3.1

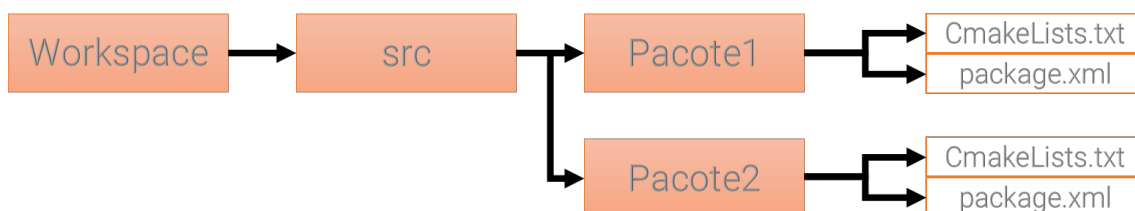


Figura 2.3.1: Estrutura Básica de Pacotes

Para criar um pacote, deve-se navegar até *<pacote>/src*.

```
$ cd ~/catkin_ws/src
```



```

10 <author email="you@yourdomain.tld">Jane Doe</author>
11
12 <buildtool_depend>catkin</buildtool_depend>
13
14 <build_depend>roscpp</build_depend>
15 <build_depend>rospy</build_depend>
16 <build_depend>std_msgs</build_depend>
17
18 <exec_depend>roscpp</exec_depend>
19 <exec_depend>rospy</exec_depend>
20 <exec_depend>std_msgs</exec_depend>
21
22 </package>

```

O arquivo *CMakeLists.txt* contém informação sobre a compilação do pacote, sendo necessário alterá-lo, basicamente, quando existir algum nó em C++. Portanto, será abordado posteriormente. Detalhes sobre isso podem ser encontrados em <http://wiki.ros.org/catkin/CMakeLists.txt>.

2.4 Navegação

O ROS provê comandos para auxiliar na navegação de dados, uma vez que as workspaces podem se encher de pacotes, arquivos e nós, dificultando encontrá-los por meio de navegações padrão. Vale lembrar que o atalho `tab` pode ser bem útil para completar os termos.

OBS: Serão usados os comandos no pacote *ros-tutorials*, portanto deve-se instalá-lo, caso ainda não tenha sido feito.

```
$ sudo apt-get install ros-kinetic-ros-tutorials
```

São basicamente três, *rospack*, *roscd* e *rosls*. O primeiro permite conseguir informações sobre o pacote. A principal informação é a localização, *rospack find <pacote>*:

```
$ rospack find roscpp
/opt/ros/kinetic/share/roscpp
```

O segundo, *roscd* serve para mudar de diretório diretamente para um pacote.

```
$ roscd roscpp
$ pwd
/opt/ros/kinetic/share/roscpp
```

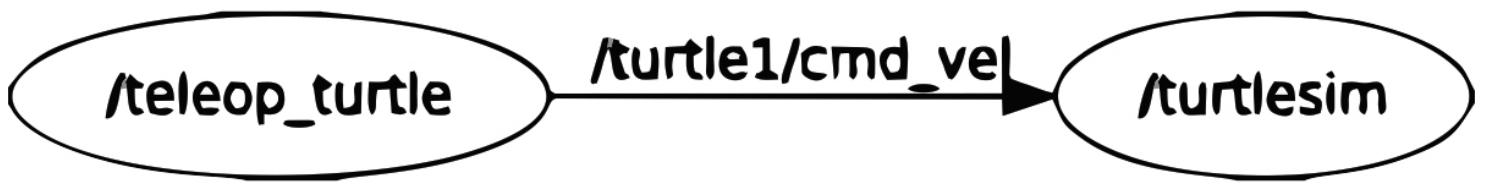
Por fim, *rosls* lista os arquivos de um pacote.

```
$ rosls roscpp_tutorials
cmake launch package.xml srv
```




Conceitos Básicos

3	Nós e Tópicos	27
4	Criando nós	35
4.1	Publicar	
4.2	Subscrever	
4.3	Compilação e Execução	
5	Hello World Turtlesim	45
5.1	Objetivo Inicial	
5.2	Refinando o método	
5.3	Alvo Variante	
6	Ferramentas	53
6.1	Serviços e Mensagens	
6.2	Parâmetros	
6.3	Servidor de Reconfiguração Dinâmica	
6.4	Remap e Namespace	
6.5	Launch	
6.6	Múltiplas Máquinas	



3. Nós e Tópicos

Como abordado anteriormente, os nós e os tópicos podem ser entendidos e representados por meio de um grafo. Cada nó é um programa que pode receber ou enviar dados para outros, e esses dados enviados, são denominados tópicos. Um mesmo programa pode receber e enviar vários tópicos, e um mesmo tópico pode ser recebido ou enviado para vários nós, não há restrição. O envio de mensagens chama-se **publicar**, e o recebimento, **subscrever**.

Para estabelecer os envios de mensagens, é necessário executar o *roscore*. Portanto, sempre que trabalhar com ROS, deve-se executar *roscore* em um primeiro terminal. Todos os outros nós comunicam com ele e assim as conexões são estabelecidas, mas para simplificar as análises, será omitido o *roscore*.

Ainda usando o pacote *ros-tutorials*:

```
# Caso ainda nao tenha sido instalado
$ sudo apt-get install ros-kinetic-ros-tutorials
```

Para executar um nó, é necessário usar o comando *roslaunch <pacote> <nó>*, e antes ter executado o *roscore* e impresso algo similar a:

```
$ roscore
... logging to /home/arthur/.ros/log/37f2df16-639b-11e9
    -9836-fc017cfe4f2b/roslaunch-arthur-linux-2416.log
Checking log directory for disk usage. This may take
    awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://arthur-linux:34225/
ros_comm version 1.12.14
```

```
SUMMARY
=====
```

```
PARAMETERS
```

```
* /rostdistro: kinetic
* /rosversion: 1.12.14
```

```
NODES
```

```
auto-starting new master
process[master]: started with pid [2563]
ROS_MASTER_URI=http://arthur-linux:11311/

setting /run_id to 37f2df16-639b-11e9-9836-fc017cfe4f2b
process[rosout-1]: started with pid [2644]
started core service [/rosout]
```

Sendo assim, o primeiro nó pode ser executado.

Em um novo terminal:

```
$ rosrund turtlesim turtlesim_node
```

Abrirá uma nova janela similar à Fig. 3.0.1

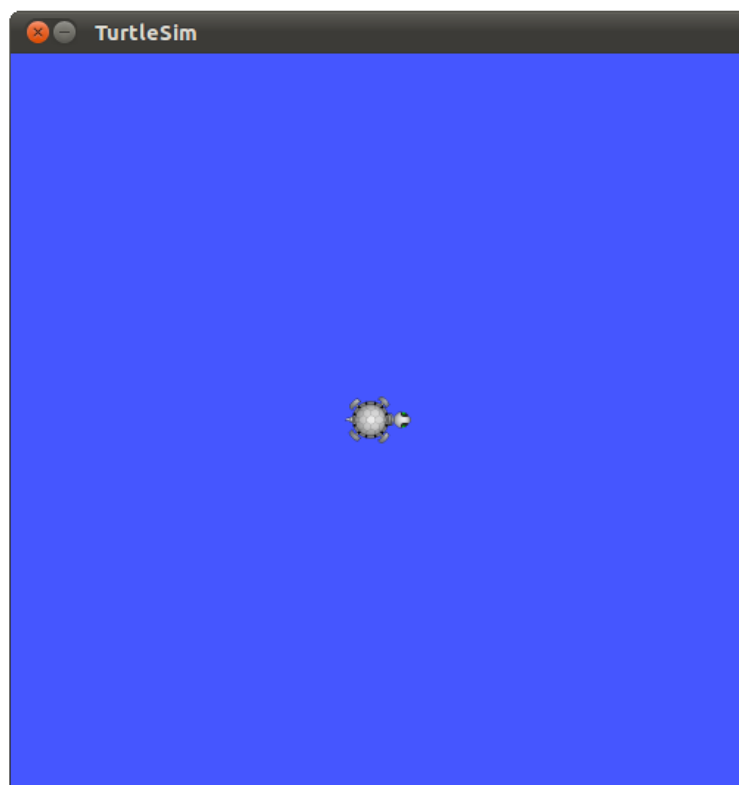


Figura 3.0.1: Primeira Execução de `turtlesim_node`

Com o sucesso da execução do nó, é possível verificá-lo pelo comando

```
$ rosnode list
/rosout
/turtlesim
```

OBS: O comando *rosnode* é uma ferramenta para adquirir informações sobre os nós.

Em outro terminal, ainda com o *roscore* e o *turtlesim_node* em execução, será executado outro nó, o *turtle_teleop_key*.

```
$ rosrund turtlesim turtle_teleop_key
[ INFO] 1254264546.878445000: Started node [/
  teleop_turtle], pid [5528], bound on [aqy], xmlrpc
  port [43918], tcpport [55936], logging to [~/ros/
  ros/log/teleop_turtle_5528.log], using [real] time
Reading from keyboard
-----
Use arrow keys to move the turtle.
```

Agora, as setas do teclado devem ser capazes de mover a tartaruga. Isso ocorre porque o nó *turtle_teleop_key* está publicando mensagens em um tópico que o nó *turtlesim_node* está subscrevendo. Para poder visualizar o grafo, o ROS possui uma ferramenta chamada *rqt_graph*. Deve ser verificada sua instalação:

```
$ sudo apt-get install ros-kinetic-rqt
$ sudo apt-get install ros-kinetic-rqt-common-plugins
```

Ela pode ser executada com um dos comandos a seguir:

```
$ rqt_graph
# OU
$ rosrund rqt_graph rqt_graph
```

Será aberta uma nova janela similar à Fig. 3.0.2:



Figura 3.0.2: Grafo de Execução dos Nós *turtlesim*

Similar ao *rosnode* para os nós, há o *rostopic* para os tópicos, e agora é possível ver os tópicos existentes com:

```
$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Outra função útil do *rostopic* é a função *echo*. Com ela, as mensagens que estão sendo publicadas no tópico são impressas no terminal. Iremos utilizá-la para acompanhar a

posição da tartaruga.

DICA : É possível ver todas as funções do *rostopic* digitando "rostopic" e pressionando tab duas vezes ou usando o comando *rostopic -h*.

Continuando com *turtlesim_node* e *turtle_teleop_key* em execução: O quadrado onde a tartaruga se move tem origem no canto inferior esquerdo e possui um tamanho de aproximadamente 11x11 unidades de posição. Pode-se utilizar o comando a seguir para verificar a posição em um novo terminal

```
$ rostopic echo /turtle1/pose
```

Mensagens com a posição serão frequentemente expostas no terminal. Elas devem se parecer com:

```
x: 5.544444561
y: 5.544444561
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
---
```

À medida em que a tartaruga se move, a posição é alterada. Em um novo terminal, pode-se usar outro *echo*, este no tópico *cmd_vel*, no qual os registros de velocidade são publicados. Nota-se, que nesse caso, novas mensagens são impressas apenas quando as setas são pressionadas, ou seja, apenas quando novas mensagens são publicadas:

```
$ rostopic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

O *rqt_graph* não atualiza automaticamente, mas ao reiniciá-lo ou ao clicar em atualizar é possível observar algo como a Fig. 3.0.3, com a adição dos *rostopic echo* usados.

O comando *rostopic* ainda pode ser executado para publicar diretamente em um tó-

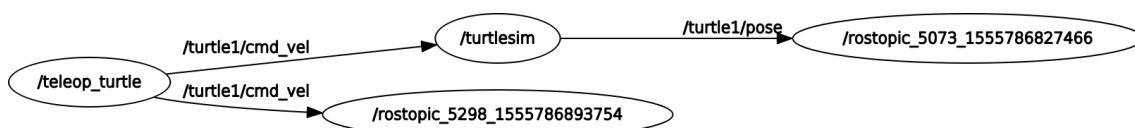


Figura 3.0.3: Grafo de Execução dos Nós *turtlesim* e *rostopic*

pico. Pode ser usado, portanto, para dar diretamente comandos de velocidades para a tartaruga. Esse procedimento necessita saber o tipo de mensagem que deve-se publicar:

```
# Verificando o tipo de mensagem
$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
# Verificando como a mensagem geometry_msgs/Twist e
  composta
$ rosmmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Feito isso, publica-se a mensagem. Lembre-se do tab.

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist
  "linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0"
```

`publishing and latching message for 3.0 seconds`

O parâmetro -1 indica que a mensagem será publicada apenas uma vez. O comando poderia ser feito sem o -1, mas o terminal travaria até que o comando seja terminado com Ctrl + C. A tartaruga deve ter feito algo como a Fig. 3.0.4.

Para postar continuamente a mensagem e fazer com que a tartaruga realize círculos, deve ser postado como:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1
  -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.0]'
```

Pode-se usar também a ferramenta *rqt_plot* para ajudar na visualização. Para isso, deve-se executá-la, digitar o que deseja visualizar no campo superior esquerdo e apertar o botão "+". Será feito com */turtle1/pose/x* e *turtle1/pose/y*.

```
$ rosrn rqt_plot rqt_plot
```

Um gráfico dinâmico com a posição será executado, e as medidas serão definidas digitando-as no espaço em branco ao clicar em "+", como a Fig. 3.0.5 mostra.

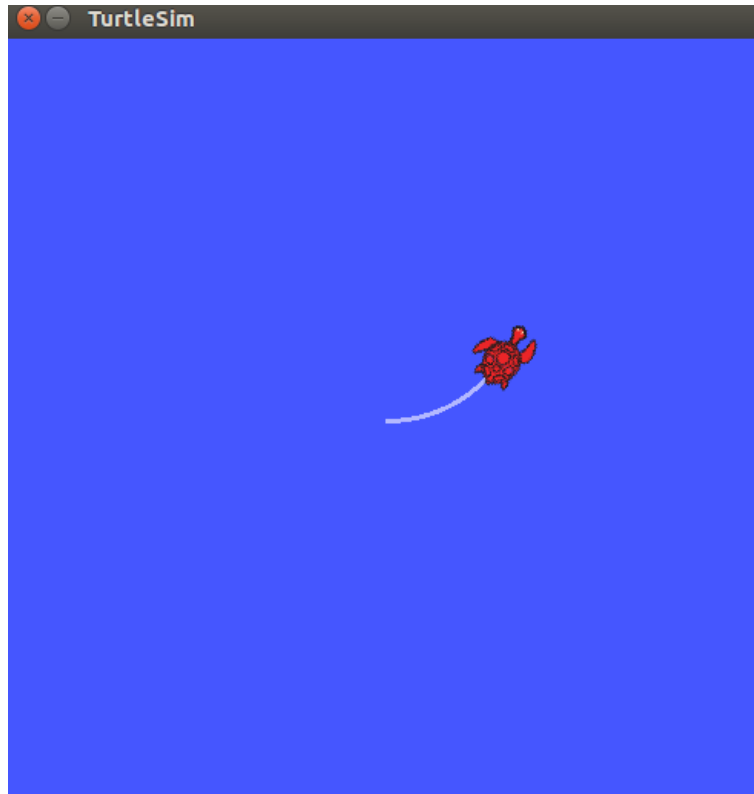


Figura 3.0.4: *Publicação direta em cmd_vel*

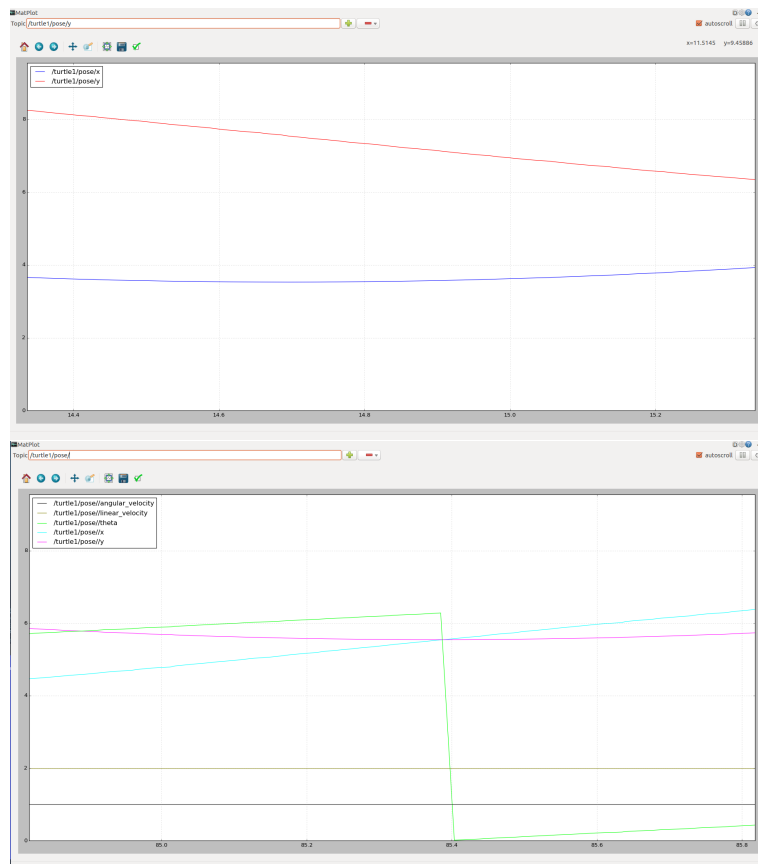


Figura 3.0.5: *Uso do rqt_plot*



4. Criando nós

Agora, nós serão criados usando os exemplos clássicos: *talker* e *listener*, o equivalente ao primeiro programa *Hello World* no ROS. Será mostrado tanto C++ quanto Python. Os códigos a seguir foram retirados de <http://wiki.ros.org/ROS/Tutorials>. É uma boa prática colocar os arquivos de Python em uma pasta chamada *scripts* e os de C++ em outra chamada *src*, ambas dentro do pacote. Por isso:

```
$ roscd begginer_tutorials
$ mkdir src
$ mkdir cripts
```

4.1 Publicar

DICA: Se fizer o código em Python, lembre-se de guardar o arquivo dentro de *begginer_tutorials/script*. Se fizer em C++, em *begginer_tutorials/src*.

A primeira linha será a mesma em todos os nós em Python. Isso serve para garantir que serão interpretados com a linguagem correta. A linha 3 também será obrigatória em todo programa para o ROS. Ela serve para importar a biblioteca *rospy*, com as devidas funções. A linha 4 é a importação do tipo de dado que será publicado, um simples contêiner de *strings*.

Da linha 6 até a 14 é a declaração da função *talker()*, a qual executa todo o procedimento de publicação, que será explicado daqui a pouco. A linha 16 é similar à declaração da função *int main* em C++. É o procedimento principal, que chama a função *talker()*, da linha 17 a 20, porém dentro de algumas estruturas. Eles garantem, sobretudo, que o procedimento da função não continue sendo executado quando o programa for desligado com Ctrl + C ou com outros meios.

Por fim, a função *talker()*. As linhas 7 e 8, que definem a interface do nó com o ROS, significam, respectivamente: que o nó irá publicar no tópico *chatter* mensagens do tipo de dado *String*, e estas mensagens, segundo o *queue_size=10*, serão acumuladas até 10, caso nenhum programa esteja recebendo as mensagens rápido o suficiente; e que o nó será iniciado com o nome *talker*, e segundo o argumento *anonymous=True*, o nó receberá um número aleatório para que seu nome seja único. A velocidade de submissão é definida pela linha frequência, na linha 9. O laço *while* possui a condição *rospy.is_shutdown()* para garantir que seja executado até que o nó seja finalizado, informa no terminal (linha 12), publica a mensagem (linha 13). E a linha 14 é para garantir a frequência desejada de publicações.

talker.py

```

1  #!/usr/bin/env python
2  # license removed for brevity
3  import rospy
4  from std_msgs.msg import String
5
6  def talker():
7      pub = rospy.Publisher('chatter', String, queue_size=10)
8      rospy.init_node('talker', anonymous=True)
9      rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass

```

Agora em C++:

A linha 1 será obrigatória para todo nó em C++. Isso serve para importarmos a biblioteca *ros/ros.h*. A segunda é para importarmos o tipo de dado que será publicado no tópico, *std_msgs/String.h*.

Todo o procedimento será contido na função *main*, que começa na linha 9. Deve-se iniciar o nó (linha 21), que precisa receber *argc* e *argv*, para permitir o remapeamento quando formos utilizá-lo, e o nome do nó.

A linha 47 informa ao *master* que o nó irá publicar mensagens do tipo *std_msgs::String* no tópico *chatter*, e enfileira até 1000 mensagens. Esse método *advertise()* vem do objeto *n* do tipo *ros::NodeHandle* (linha 28) e serve para o propósito de conter o método que publica no tópico e automaticamente desligá-lo quando estiver fora de escopo. A frequência de publicação é definida pela linha 49.

A condição do *while* da linha 56 irá retornar falso se o programa for desligado com Ctrl + C, se o nó for expulso da rede por outro com o mesmo nome, se *ros::shutdown()* for chamado por outra parte da aplicação ou se todos os *ros::NodeHandles* forem destruídos. O resto do laço é responsável pela publicação. A linha 67 é responsável por imprimir no terminal a mensagem, um tipo de *print*. A linha 77 não é necessária neste nó, seria útil apenas se fosse subscrever em algum outro tópico. A linha 79 é para garantir a frequência de publicação desejada.

talker.cpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 /**
7  * This tutorial demonstrates simple sending of messages
8  * over the ROS system.
9  */
10 int main(int argc, char **argv)
11 {
12     /**
13      * The ros::init() function needs to see argc and argv
14      * so that it can perform
15      * any ROS arguments and name remapping that were
16      * provided at the command line.
17      * For programmatic remappings you can use a different
18      * version of init() which takes
19      * remappings directly, but for most command-line
20      * programs, passing argc and argv is
21      * the easiest way to do it. The third argument to
22      * init() is the name of the node.
23      *
24      * You must call one of the versions of ros::init()
25      * before using any other
26      * part of the ROS system.
27      */
28     ros::init(argc, argv, "talker");
29
30     /**
31      * NodeHandle is the main access point to
32      * communications with the ROS system.
33      * The first NodeHandle constructed will fully
34      * initialize this node, and the last
35      * NodeHandle destructed will close down the node.
36      */
37     ros::NodeHandle n;
```

```
30  /**
31   * The advertise() function is how you tell ROS that
      you want to
32   * publish on a given topic name. This invokes a call
      to the ROS
33   * master node, which keeps a registry of who is
      publishing and who
34   * is subscribing. After this advertise() call is made,
      the master
35   * node will notify anyone who is trying to subscribe
      to this topic name,
36   * and they will in turn negotiate a peer-to-peer
      connection with this
37   * node. advertise() returns a Publisher object which
      allows you to
38   * publish messages on that topic through a call to
      publish(). Once
39   * all copies of the returned Publisher object are
      destroyed, the topic
40   * will be automatically unadvertised.
41   *
42   * The second parameter to advertise() is the size of
      the message queue
43   * used for publishing messages. If messages are
      published more quickly
44   * than we can send them, the number here specifies how
      many messages to
45   * buffer up before throwing some away.
46   */
47  ros::Publisher chatter_pub = n.advertise<std_msgs::
      String>("chatter", 1000);
48
49  ros::Rate loop_rate(10);
50
51  /**
52   * A count of how many messages we have sent. This is
      used to create
53   * a unique string for each message.
54   */
55  int count = 0;
56  while (ros::ok())
57  {
58      /**
59       * This is a message object. You stuff it with data,
          and then publish it.
60       */
61      std_msgs::String msg;
62
```

```

63     std::stringstream ss;
64     ss << "hello world " << count;
65     msg.data = ss.str();
66
67     ROS_INFO("%s", msg.data.c_str());
68
69     /**
70      * The publish() function is how you send messages.
71      * The parameter
72      * is the message object. The type of this object
73      * must agree with the type
74      * given as a template parameter to the advertise <>()
75      * call, as was done
76      * in the constructor above.
77      */
78     chatter_pub.publish(msg);
79
80     ros::spinOnce();
81
82     loop_rate.sleep();
83     ++count;
84 }
85
86 return 0;
87 }

```

4.2 Subscreever

Neste caso, temos duas funções declaradas, a que realiza os procedimentos de início de nó *listener()*, similar à *talker()*, e uma extra, que realiza as atribuições da subscrição, geralmente chamada de *callback*.

As três primeiras linhas têm o mesmo objetivo das três primeiras linhas de *talker.py*, garantir que seja interpretado por Python, importar a biblioteca *rospy* e importar o tipo de dado que irá ser usado (neste caso, para subscreever).

A linha 15 é a inicialização do nó com argumento *anonymous=True* para garantir a unicidade do nome. A linha 17 declara que o nó irá subscreever no tópico *chatter*, recebendo o tipo de dado *String* e chamando a função *callback*, com a mensagem sendo seu primeiro argumento. A linha 20 impede que o nó termine até que seja desligado. Ao contrário de C++, o comando *spin* não afeta as funções de *callback*, pois possuem seus próprios encadeamentos.

Por fim, a função *callback*, que é executada sempre que há uma publicação no tópico, imprime uma mensagem no terminal. Ela é declarada como uma *thread*, logo não precisa ser sempre executada.

listener.py

```

1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import String
4
5 def callback(data):
6     rospy.loginfo(rospy.get_caller_id() + "I heard %s",
7                 data.data)
8
9 def listener():
10     # In ROS, nodes are uniquely named. If two nodes with
11     # the same
12     # name are launched, the previous one is kicked off.
13     # The
14     # anonymous=True flag means that rospy will choose a
15     # unique
16     # name for our 'listener' node so that multiple
17     # listeners can
18     # run simultaneously.
19     rospy.init_node('listener', anonymous=True)
20     rospy.Subscriber("chatter", String, callback)
21
22     # spin() simply keeps python from exiting until this
23     # node is stopped
24     rospy.spin()
25
26 if __name__ == '__main__':
27     listener()

```

Agora em C++:

Analogamente ao caso de Python, as declarações iniciais são as mesmas e há uma função a mais do que *talker.cpp*, neste programa, chamada de *chatterCallback*.

A linha 24 inicializa o nó, e o objeto sub (linha 48) é o método *subscribe()* de um *ros::NodeHandle*. Esse método indica que subscreverá no tópico *chatter*, com um tamanho de fila de 1000 e chamará a função *chatterCallback* sempre que uma nova mensagem for publicada.

A linha 55 irá garantir a chamada de callbacks o mais rápido possível, até que *ros::ok()* retorne falso.

listener.cpp

```

1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3

```



```
4 /**
5  * This tutorial demonstrates simple receipt of messages
6  * over the ROS system.
7  */
8 void chatterCallback(const std_msgs::String::ConstPtr&
9  msg)
10 {
11   ROS_INFO("I heard: [%s]", msg->data.c_str());
12 }
13
14 int main(int argc, char **argv)
15 {
16   /**
17    * The ros::init() function needs to see argc and argv
18    * so that it can perform
19    * any ROS arguments and name remapping that were
20    * provided at the command line.
21    * For programmatic remappings you can use a different
22    * version of init() which takes
23    * remappings directly, but for most command-line
24    * programs, passing argc and argv is
25    * the easiest way to do it. The third argument to
26    * init() is the name of the node.
27    *
28    * You must call one of the versions of ros::init()
29    * before using any other
30    * part of the ROS system.
31    */
32   ros::init(argc, argv, "listener");
33
34   /**
35    * NodeHandle is the main access point to
36    * communications with the ROS system.
37    * The first NodeHandle constructed will fully
38    * initialize this node, and the last
39    * NodeHandle destructed will close down the node.
40    */
41   ros::NodeHandle n;
42
43   /**
44    * The subscribe() call is how you tell ROS that you
45    * want to receive messages
46    * on a given topic. This invokes a call to the ROS
47    * master node, which keeps a registry of who is
48    * publishing and who
49    * is subscribing. Messages are passed to a callback
50    * function, here
51    * called chatterCallback. subscribe() returns a
```

```

    Subscriber object that you
39  * must hold on to until you want to unsubscribe. When
    all copies of the Subscriber
40  * object go out of scope, this callback will
    automatically be unsubscribed from
41  * this topic.
42  *
43  * The second parameter to the subscribe() function is
    the size of the message
44  * queue. If messages are arriving faster than they
    are being processed, this
45  * is the number of messages that will be buffered up
    before beginning to throw
46  * away the oldest ones.
47  */
48  ros::Subscriber sub = n.subscribe("chatter", 1000,
    chatterCallback);
49
50  /**
51  * ros::spin() will enter a loop, pumping callbacks.
    With this version, all
52  * callbacks will be called from within this thread (
    the main one). ros::spin()
53  * will exit when Ctrl-C is pressed, or the node is
    shutdown by the master.
54  */
55  ros::spin();
56
57  return 0;
58  }

```

4.3 Compilação e Execução

Criados os códigos, agora eles serão executados. Para os nós gerados em Python, é preciso conceder a permissão de execução, que pode ser dada da seguinte forma:

```

$ roscd begginer_tutorials/scripts
$ chmod +x talker.py
$ chmod +x listener.py
$ cd ~/catkin_ws
# Nao e necessario compilar a workspace depois de se
  fazer/editar scripts em python
# catkin_make

```

É recomendado compilar a *workspace* com os dois últimos comandos, mas não é necessário fazê-lo, principalmente se alterações forem feitas constantemente nos códigos. Uma vez concebida a permissão, ele será executado, mesmo que modificado.

Já para os nós criados em C++, é preciso alterar o arquivo *CMakeLists.txt* para que

seja compilado, adicionando as seis últimas linhas no arquivo. Desta vez é preciso recomilar com *catkin_make* sempre que o código for modificado.

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy
             std_msgs genmsg)

## Declare ROS messages and services
add_message_files(FILES Num.msg)
add_service_files(FILES AddTwoInts.srv)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package()

## Build talker and listener
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker
                 beginner_tutorials_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener
                 beginner_tutorials_generate_messages_cpp)
```

No terminal:

```
$ cd ~/catkin_ws
$ catkin_make
```

Agora pode-se executar os nós (lembrando que é preciso dar *roscore* e *source* da workspace).

Primeiro, os de em Python:

```
$ cd ~/catkin_ws
# Lembre-se de dar roscore em um outro terminal e source
  devel/setup.bash em todos os novos terminais
$ rosrn beginner_tutorials talker.py
[INFO] [1555872563.477477]: hello world 1555872563.48
[INFO] [1555872563.577758]: hello world 1555872563.58
```

Em um novo terminal:

```
$ rosrun beginner_tutorials listener.py
[INFO] [1555872598.581874]: /listener_8750_1555872598310I
      heard hello world 1555872598.58
[INFO] [1555872598.682315]: /listener_8750_1555872598310I
      heard hello world 1555872598.68
```

Significa que as mensagens estão sendo publicadas pelo *talker.py* e lidas pelo *listener.py*.

Em segundo, fechando os nós de Python (Ctrl + C), pode-se fazer o mesmo para os gerados em C++:

```
$ cd ~/catkin_ws
# Lembre-se de dar roscore em um outro terminal e source
  devel/setup.bash em todos os novos terminais
$ rosrun beginner_tutorials talker
# Em um novo terminal
$ rosrun beginner_tutorials listener
```

Pode-se, inclusive, executar um em C++ e outro em Python, ou até mesmo os quatro de uma só vez. Executando-os e abrindo a ferramenta *rqt_graph* para melhor visualização:

```
$ cd ~/catkin_ws
# Lembre-se de dar roscore em um outro terminal e source
  devel/setup.bash em todos os novos terminais
$ rosrun beginner_tutorials talker
# Em um novo terminal
$ rosrun beginner_tutorials listener
# Em um novo terminal
$ rosrun beginner_tutorials talker
# Em um novo terminal
$ rosrun beginner_tutorials listener
# Em um novo terminal
$ rosrun rqt_graph rqt_graph
```

No canto superior esquerdo de *rqt_graph*, pode-se selecionar a opção "Nodes/Topics (all)", na qual os nós são representados pelos círculos e os tópicos pelos quadrados, e visualizar algo como a Fig. 4.3.1.

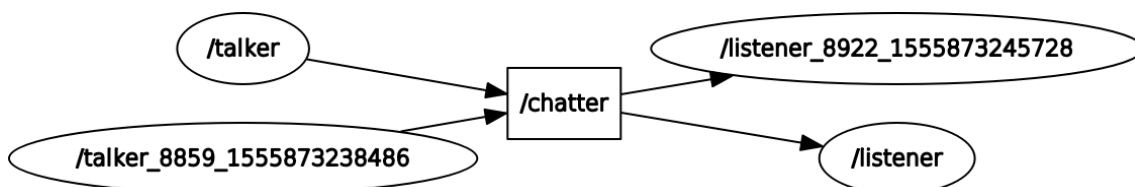
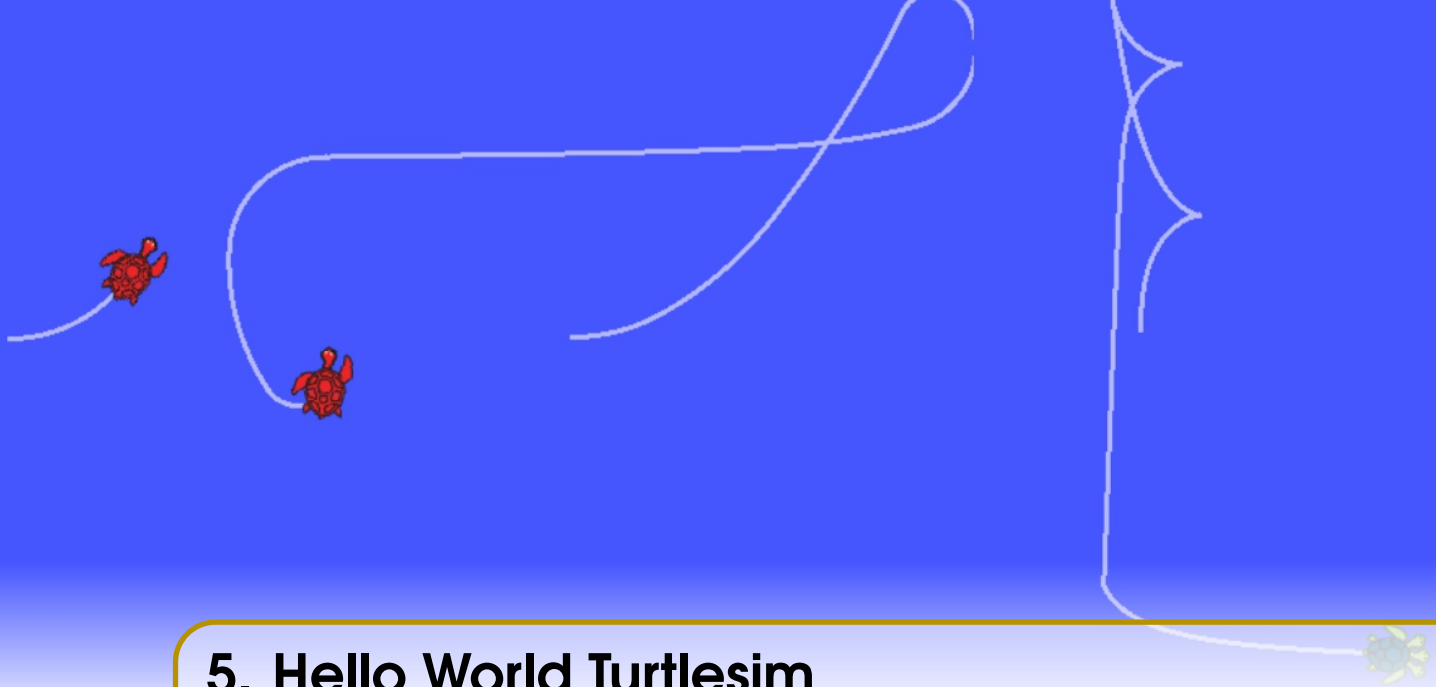


Figura 4.3.1: Grafo de dois talker e dois listener



5. Hello World Turtlesim

Para fixar a ideia de construção de nós, agora serão criados outros, em Python, para controle da tartaruga *turtlesim* a fim de fazer uma exemplificação prática.

CURIOSIDADE : O *turtlesim* é conhecido como o Hello World do ROS, isto é, o primeiro programa e o primeiro exemplo utilizado para o seu aprendizado.

5.1 Objetivo Inicial

O primeiro objetivo é controlar a tartaruga de um ponto inicial qualquer para outro ponto alvo dentro de seu *workspace*. Para isso, será necessário publicar dados de velocidade e subscrever dados da posição da tartaruga. Esses dados encontram-se, respectivamente, nos tópicos */turtle1/cmd_vel* e */turtle1/pose*. Isso pode ser verificado executando o nó *turtlesim_node*, verificando a lista de tópicos com *rostopic list* e executando *rostopic echo* para ambos os tópicos.

Para o comando de velocidade, precisa-se da velocidade linear e angular, e os dados de posição que necessitamos são x , y e θ .

Sendo assim, as velocidades linear (V) e angular (ω) serão calculadas como:

$$V = K\sqrt{(\Delta Y)^2 + (\Delta X)^2}$$

$$\omega = \text{atan2}(\Delta Y, \Delta X)$$

$$\Delta Y = y_{\text{alvo}} - y_{\text{atual}}, \Delta X = x_{\text{alvo}} - x_{\text{atual}}$$

O procedimento será realizado até que a tartaruga esteja próxima do ponto, definido por:

$$|\Delta Y| < \varepsilon, |\Delta X| < \varepsilon$$

ε = erro máximo admitido. Sendo assim, basta implementar o código:

controle_vel.py

```

1  #!/usr/bin/env python
2  import rospy
3  from geometry_msgs.msg import Twist
4  from turtlesim.msg import Pose
5  from math import atan2, sqrt
6  x0 = 0.0
7  y0 = 0.0
8  xf = 0.0
9  yf = 0.0
10 Theta0 = 0.0
11 Err = 0.2
12 Gain = 1.0
13 d = 0.2
14 def atribuirvalorpos(data):
15     global x0
16     global y0
17     global Theta0
18     x0 = data.x
19     y0 = data.y
20     Theta0 = data.theta
21 def mover():
22     global xf, yf, Err, Gain, d
23     rospy.init_node('controle_vel', anonymous=True)
24     pub = rospy.Publisher('turtle1/cmd_vel', Twist,
25         queue_size=1)
26     rate = rospy.Rate(1)
27     vel_msg = Twist()
28     rospy.Subscriber('turtle1/pose', Pose, atribuirvalorpos
29         )
30     while not rospy.is_shutdown():
31         xf, yf = raw_input('Digite a coordenada (x,y)
32             desejada: ').split()
33         xf, yf = [float(xf) for xf in [xf, yf]]
34         if xf > 11 or xf < 0:
35             rospy.loginfo('\033[31mCoordenadas invalidas,
36                 digite novamente...\033[m')
37             continue
38         if yf > 11 or yf < 0:
39             rospy.loginfo('\033[31mCoordenadas invalidas,
40                 digite novamente...\033[m')
41             continue
42         while abs(x0-xf) > Err or abs(y0-yf) > Err:
43             vel_msg.angular.z = atan2((yf-y0),(xf-x0)) -
44                 Theta0

```

```

40         vel_msg.linear.x = Gain * sqrt (( (xf-x0)**2 +
        (yf-y0)**2 ))
41         if vel_msg.linear.x > 2:
42             vel_msg.linear.x = 2
43         elif vel_msg.linear.x < -2:
44             vel_msg.linear.x = -2
45         pub.publish(vel_msg)
46         rate.sleep()
47 if __name__ == '__main__':
48     try:
49         mover()
50     except rospy.ROSInterruptException:
51         pass

```

As cinco primeiras linhas são o cabeçalho do código, garantindo que seja interpretado por Python e importando as bibliotecas, métodos e tipos de dados necessários.

A função *atribuirvalores* é a função *callback*, que serve para armazenar os dados da posição em variáveis. A função *mover* contém o procedimento principal. Nela o nó é iniciado (linha 23) e as declarações de publicar no tópico de velocidade (linha 24) e subscrever no tópico de posição (linha 27) são feitas.

Enquanto o programa não for desligado (linha 29), o nó recebe os valores de posição desejada do usuário (linhas 30 e 31), confere se são válidos (linha 32 até 37), e realiza um laço até que a posição em que tartaruga está esteja em uma boa proximidade do ponto desejado (linha 38).

Dentro desse laço, os cálculos são feitos e a mensagem com as velocidades linear e angular desejadas são publicadas. Da linha 41 até 44 há limitações pouco elegantes da velocidade linear para que a tartaruga não se mova muito rápido para frente.

OBS: Os caracteres "\033[m" nas linhas 33 e 36 indicam cores para as mensagens impressas em Python.

Com isso, o programa é executado e um resultado é mostrado na Fig. 5.1.1:

```

$ cd ~/catkin_ws/src/beginner_tutorials/scripts/
$ chmod +x controle_vel.py
$ cd ~/catkin_ws
$ rosrn turtlesim turtlesim_node
# Novo terminal
# source devel/setup.bash
$ rosrn beginner_tutorials controle_vel.py
Digite a coordenada (x,y) desejada: 10 10
Digite a coordenada (x,y) desejada: 2 8
Digite a coordenada (x,y) desejada: 2 5
Digite a coordenada (x,y) desejada:

```



Figura 5.1.1: Movimento da tartaruga

5.2 Refinando o método

O objetivo de controlar a tartaruga até um ponto final continua, porém agora objetiva-se também utilizar uma linearização por realimentação de estados estática. Ela visa controlar o movimento da ponta de seu robô, que está uma distância d do centro. Sendo assim, o cálculo da velocidade é feito da seguinte forma:

$$\begin{pmatrix} V \\ \omega \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\frac{\sin(\theta)}{d} & \frac{\cos(\theta)}{d} \end{pmatrix} \times \begin{pmatrix} U_x \\ U_y \end{pmatrix}$$

$$\begin{pmatrix} U_x \\ U_y \end{pmatrix} = U = V_{ref} + KP$$

$$V_{ref} = \begin{pmatrix} V_{refx} \\ V_{refy} \end{pmatrix}, P = \begin{pmatrix} \Delta X \\ \Delta Y \end{pmatrix}$$

V_{ref} é a velocidade do ponto alvo, não interessando por agora, uma vez que o alvo é estático, isto é, invariante com o tempo.

Implementando o código, de maneira análoga ao anterior, apenas alterando o cálculo da velocidade:

controle_vel2.py

```
1 #!/usr/bin/env python
```



```
2 import rospy
3 from turtlesim.msg import Pose
4 from geometry_msgs.msg import Twist
5 from math import cos, sin
6 x0 = 0.0
7 y0 = 0.0
8 T = 0.0
9 k = 1
10 d = 1
11 Err = 0.1
12 vel_lim = Twist()
13 vel_lim.linear.x = 3
14 vel_lim.angular.z = 3
15 def callback(data):
16     global x0, y0, T, k, d
17     x0 = data.x + d * cos(data.theta)
18     y0 = data.y + d * sin(data.theta)
19     T = data.theta
20 def talker():
21     rospy.init_node('controle_vel', anonymous=True)
22     pub = rospy.Publisher('turtle1/cmd_vel', Twist,
23         queue_size=1)
24     rospy.Subscriber('/turtle1/pose', Pose, callback)
25     rate = rospy.Rate(20)
26     vel_msg = Twist()
27     while not rospy.is_shutdown():
28         xf, yf = raw_input('Digite a coordenada (x,y)
29             desejada: ').split()
30         xf, yf = [float(xf) for xf in [xf, yf]]
31         if xf > 11 or xf < 0 or yf > 11 or yf < 0:
32             rospy.loginfo('\033[31mCoordenadas invalidas,
33                 digite novamente...\033[m')
34             continue
35         lim = 0
36         while abs(xf - x0) > Err or abs(yf - y0) > Err:
37             vel_msg.linear.x = k * ( cos(T) * (xf - x0) +
38                 sin(T) * (yf - y0) )
39             vel_msg.angular.z = k * (-(sin(T) * (xf - x0))
40                 / d + (cos(T) * (yf - y0)) / d)
41             if vel_msg.linear.x > vel_lim.linear.x:
42                 vel_msg.linear.x = vel_lim.linear.x
43             elif vel_msg.linear.x < -vel_lim.linear.x:
44                 vel_msg.linear.x = -vel_lim.linear.x
45             if vel_msg.angular.z > vel_lim.angular.z:
46                 vel_msg.angular.z = vel_lim.angular.z
47             elif vel_msg.angular.z < -vel_lim.angular.z:
48                 vel_msg.angular.z = -vel_lim.angular.z
49         pub.publish(vel_msg)
```

```

45         rate.sleep()
46 if __name__ == '__main__':
47     try:
48         talker()
49     except rospy.ROSInterruptException:
50         pass

```

A velocidade ainda não está sendo limitada de uma boa forma, mas por enquanto servirá.

Ao ser executado:

```

$ rosrun beginner_tutorials controle_vel6.py
Digite a coordenada (x,y) desejada: 0 5
Digite a coordenada (x,y) desejada: 10 5
Digite a coordenada (x,y) desejada: 10 10
Digite a coordenada (x,y) desejada:

```

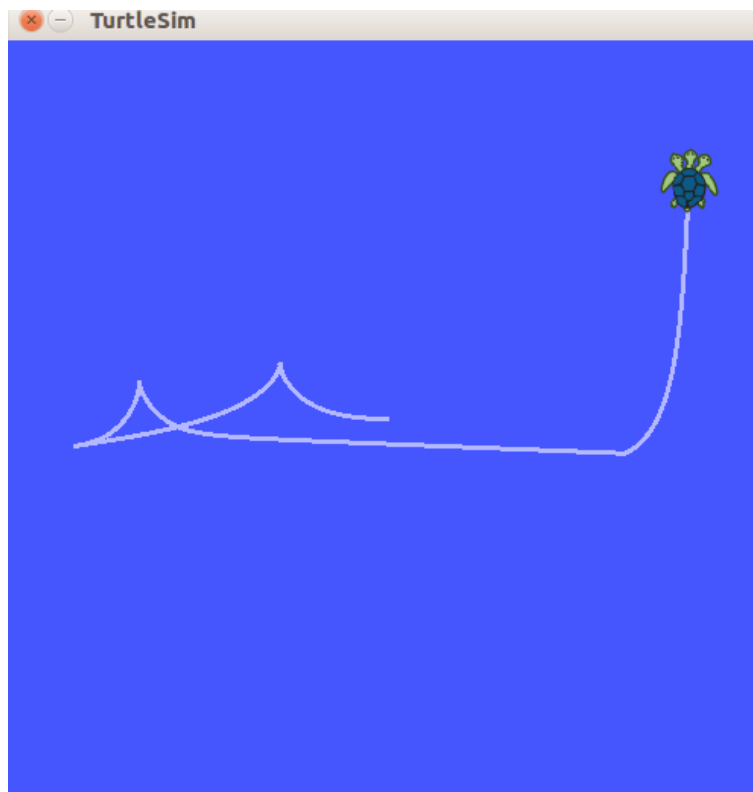


Figura 5.2.1: Movimento da tartaruga

5.3 Alvo Variante

Deseja-se, agora, manter os dois objetivos anteriores e fazer com que o ponto alvo varie com o tempo. Assim, a tartaruga perseguirá uma curva. Desta forma, o V_{ref} será diferente de zero. Para isso, precisa-se definir a expressão para o ponto alvo. Será utilizado uma elipse paramétrica como exemplo:

$$x_f = a \cos(wt) + c_x$$

$$y_f = b \sin(\omega t) + c_y$$

Onde a e b são os semieixos e c é o centro.

Sendo assim, V_{ref} será a velocidade desse ponto, que é a derivada deles em relação ao tempo:

$$V_{ref} = \begin{pmatrix} V_{refx} \\ V_{refy} \end{pmatrix} = \begin{pmatrix} \frac{\partial x_f}{\partial t} \\ \frac{\partial y_f}{\partial t} \end{pmatrix} = \begin{pmatrix} -a \sin(\omega t) \omega \\ b \cos(\omega t) \omega \end{pmatrix}$$

Por fim, falta alterar os cálculos e adicionar uma variável tempo, incrementada de acordo com 1/frequência.

CURIOSIDADE: Este exemplo é uma simplificação de um método de controle de *trajectory tracking*, chamado de *Feedback Linearization*.

controle_elipse.py

```

1 #!/usr/bin/env python
2 import rospy
3 from turtlesim.msg import Pose
4 from geometry_msgs.msg import Twist
5 from math import cos, sin
6 x0 = 0.0
7 y0 = 0.0
8 T = 0.0
9 k = 0.2
10 d = 1
11 def callback(data):
12     global x0, y0, T, k, d
13     x0 = data.x + d * cos(data.theta)
14     y0 = data.y + d * sin(data.theta)
15     T = data.theta
16 def iniciar():
17     rospy.init_node('controle_vel', anonymous=True)
18     pub = rospy.Publisher('turtle1/cmd_vel', Twist,
19                             queue_size=1)
19     rospy.Subscriber('/turtle1/pose', Pose, callback)
20     rate = rospy.Rate(20)
21     vel_msg = Twist()
22     cx, cy, rx, ry = raw_input('Digite o centro (x,y) e os
23                             semieixos (a,b) ').split()
23     cx, cy, rx, ry = [float(i) for i in [cx, cy, rx, ry]]
24     t = 0
25     w = 0.4
26     while not rospy.is_shutdown():
27         t += 0.05
28         xf = rx * cos(w*t) + cx

```

```

29     yf = ry * sin(w*t) + cy
30     Vx = k * (xf - x0) - rx*sin(w*t)*w
31     Vy = k * (yf - y0) + ry*cos(w*t)*w
32     vel_msg.linear.x = cos(T) * Vx + sin(T) * Vy
33     vel_msg.angular.z = -(sin(T) * Vx) / d + (cos(T) *
        Vy) / d
34     rate.sleep()
35     pub.publish(vel_msg)
36 if __name__ == '__main__':
37     try:
38         iniciar()
39     except rospy.ROSInterruptException:
40         pass

```

Executamos e observamos o resultado:

```

$ rosrund beginner_tutorials controle_elipse.py
Digite o centro (x,y) e os semieixos (a,b) 4 5 2 3

```

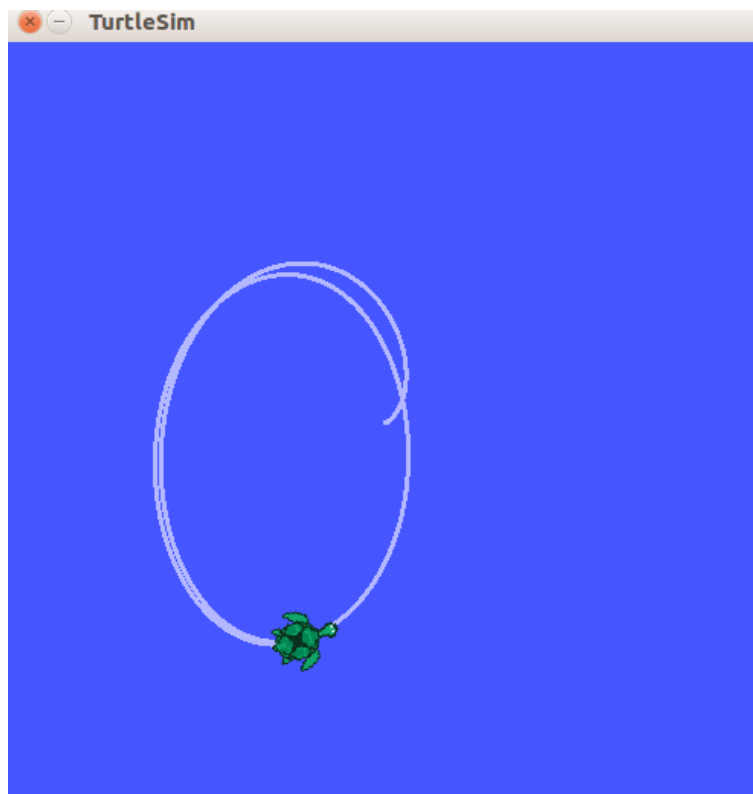


Figura 5.3.1: Movimento da tartaruga



6. Ferramentas

6.1 Serviços e Mensagens

Serviços são uma outra maneira com a qual os nós podem se comunicar entre si, enquanto uma mensagem é o "protocolo" através do qual um tópico realiza a comunicação entre dois nós. Um serviço permite que um dado nó envie requisições (requests) e receba respostas (responses). Ficará mais claro no decorrer deste capítulo a importância de se entender, e utilizar, os diferentes tipos de serviços e mensagens no ROS.

6.1.1 Entendendo Arquivos `.msg` e `.srv`

Em um primeiro momento, será abordado como criar tipos customizados de mensagens e serviços, pois a estrutura dos arquivos dos tipos `.msg` e `.srv` é o que define como os nós se comunicam.

Arquivos `.msg` são arquivos de texto que descrevem os campos de uma mensagem ROS, e as mensagens, por sua vez, podem ser compreendidas como o tipo de informação que um tópico é capaz de transmitir. Já os arquivos `.srv` descrevem os campos de um serviço ROS, sendo composto por duas partes: uma requisição (request), que são as informações que o nó de serviço recebe para processar, e uma resposta (response), que pode ser entendida como o produto final desse processamento.

Arquivos `.msg` são armazenados no diretório `msg` do pacote, assim como os arquivos `.srv` são armazenados no diretório `srv`.

Os `.msg` são simples arquivos de texto contendo um campo de tipo e um campo de nome por linha. Os tipos podem ser:

- `int8`, `int16`, `int32`, `int64` (além de `uint*`)
- `float32`, `float64`

- `string`
- `time`, `duration`
- `array[]` de tamanho variável, ou `array[C]` de tamanho fixo `C`
- outros arquivos `.msg`
- Há também um tipo especial `Header`, que contém informações de timestamp e das coordenadas do frame que são comumente usados no ROS. Frequentemente, a primeira linha dos arquivos `.msg` possui: `Header header`.

Segue abaixo um exemplo de uma mensagem que usa um `Header`, um tipo `string` primitivo e mais duas outras mensagens:

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Arquivos `.srv` são similares aos arquivos `.msg`, com a diferença de que suas duas partes são separadas por uma linha com `'—'`, mas continuam valendo a mesma estrutura de campo de tipo seguido por campo de nome, e os mesmos tipos. Segue abaixo um exemplo de um arquivo `.srv`:

```
int64 A
int64 B
——
int64 Sum
```

No exemplo acima, `A` e `B` pertencem à requisição e `Sum` pertence à resposta.

Mas antes de usar os arquivos como os exemplificados acima para definir novos tipos de mensagens e serviços, é necessário preparar o pacote.

6.1.2 Preparação

Primeiro, defini-se uma nova mensagem dentro do pacote.

```
$ cd ~/catkin_ws/src/beginner_tutorials
$ mkdir msg
$ echo "int64 num" > msg/Num.msg
```

Com um arquivo `.msg` criado, como o do exemplo acima, é necessário que ele tenha sido gerado em código-fonte para C++, Python e outras linguagens. Para fazer essa verificação, pode-se abrir o `package.xml`, localizar estas duas linhas estão inclusas e descomentá-las:

```
1 <build_depend>message_generation </build_depend>
2 <run_depend>message_runtime </run_depend>
```

Em seguida, deve-se abrir o `CMakeLists.txt` e adicionar a dependência `message_generation` na chamada `find_package`, que já existe no arquivo `CMakeLists.txt` para que se possa gerar novas mensagens. É possível fazer isso adicionando `"message_generation"` na lista `"REQUIRED COMPONENTS"`, ficando da seguinte forma:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
```

```
std_msgs
message_generation
)
```

Depois, exporta-se a dependência *message_runtime*:

```
catkin_package(
    ...
    CATKIN_DEPENDS message_runtime ...
...)
```

Agora, deve-se localizar o seguinte bloco de código:

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

Deve-se descomentar as linhas e trocar "Message1.msg" e "Message2.msg" pelo nome do arquivo *.msg*. Ficando da seguinte forma:

```
add_message_files(
    FILES
    Num.msg
)
```

Agora é necessário confirmar que a função *generate_messages()* é chamada. Para isso, é necessário localizar e descomentar as seguintes linhas:

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

Feito todos esses passos basta compilar o *workspace* e o tipo customizado de mensagem já poderá ser usado.

Depois de ter completado todas as alterações anteriores no CMakeLists.txt fica mais fácil utilizar um tipo customizado de serviço, pois há vários passos em comum. Então, indo direto ao ponto e criando um *.srv*:

```
$ roscd beginner_tutorials
$ mkdir srv
$ roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts
.srv
```

A última linha copia um tipo de serviço definido no pacote *rospy_tutorials* para o pacote *beginner_tutorials*.

Agora deve-se abrir o CMakeLists.txt e encontrar o seguinte bloco de código:

```
# add_service_files(
#   FILES
#   Service1.srv
```

```
# Service2.srv
# )
```

Descomentando as linhas e trocando "Service1.srv" e "Service2.srv" pelo nome do arquivo `.srv`, Fica da seguinte maneira:

```
add_service_files (
    FILES
    AddTwoInts.srv
)
```

Depois de compilar o *workspace*, já é possível utilizar o tipo customizado de serviço.

Desta forma, a próxima subseção irá cobrir utilização de tipos customizado de serviços. A utilização de tipos customizados de mensagens é da mesma forma que a utilização dos tipos padrões.

6.1.3 Utilização de `.srv`

DICA: Lembre-se de guardar os códigos em Python dentro de *beginner_tutorials/script*, e os códigos em C++ em *beginner_tutorials/src*.

É preciso muito pouco para escrever um serviço usando `rospy`. O nó é declarado utilizando `init_node()`, como na linha 11, e então o serviço é declarado. Na linha 12 é declarado um novo serviço do tipo `AddTwoInts` chamado `add_two_ints`, e todas as requisições são passadas para a função `handle_add_two_ints`. A `handle_add_two_ints` é chamada com instâncias de `AddTwoIntsRequest` e retorna instâncias de `AddTwoIntsResponse`.

O método `rospy.spin()` na linha 14 evita que o código termine de executar até que o serviço seja finalizado.

`add_two_ints_server.py`

```
1 #!/usr/bin/env python
2
3 from beginner_tutorials.srv import AddTwoInts,
   AddTwoIntsResponse
4 import rospy
5
6 def handle_add_two_ints(req):
7     print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a
   + req.b))
8     return AddTwoIntsResponse(req.a + req.b)
9
10 def add_two_ints_server():
11     rospy.init_node('add_two_ints_server')
12     s = rospy.Service('add_two_ints', AddTwoInts,
   handle_add_two_ints)
13     print "Ready to add two ints."
14     rospy.spin()
15
```



```

16 if __name__ == "__main__":
17     add_two_ints_server()

```

DICA :Não se esqueça de dar permissão de execução ao nó:

```
$ chmod +x scripts/add_two_ints_server.py
```

Agora que o *script* que fornece o serviço já foi criado, um outro será programado para ser o cliente.

O código do cliente para chamar serviços também é simples. Nos clientes, não é preciso chamar *init_node()*, no lugar disso é chamado o método *wait_for_service()*, como na linha 8. Esse é um método conveniente que bloqueia o código até que o serviço esperado esteja disponível. É interessante utilizar um sistema de *timeout*, para que caso o serviço esteja indisponível durante muito tempo o usuário seja notificado e possa checar se o serviço está de fato sendo executado.

Na linha 10 a variável *add_two_ints* recebe a função *handle* do serviço "add_two_ints" que é do tipo *AddTwoInts*. Na linha 11 esse *handle* é utilizado tendo como parâmetros as variáveis *x* e *y*, e na linha 12 a resposta do serviço, *resp1.sum*, é retornada. Como o tipo do serviço foi declarado como sendo *AddTwoInts*, ele faz o trabalho de gerar o objeto *AddTwoIntsRequest*. O valor de retorno é um objeto *AddTwoIntsResponse*. Se a chamada falhar, uma exceção *rospy.ServiceException* será lançada. Sendo assim é interessante também escrever um bloco *try/except* para tratar essa exceção.

add_two_ints_client.py

```

1  #!/usr/bin/env python
2
3  import sys
4  import rospy
5  from beginner_tutorials.srv import *
6
7  def add_two_ints_client(x, y):
8      rospy.wait_for_service('add_two_ints')
9      try:
10         add_two_ints = rospy.ServiceProxy('add_two_ints',
11             AddTwoInts)
12         resp1 = add_two_ints(x, y)
13         return resp1.sum
14     except rospy.ServiceException, e:
15         print "Service call failed: %s"%e
16
17 def usage():
18     return "%s [x y]"%sys.argv[0]
19
20 if __name__ == "__main__":
21     if len(sys.argv) == 3:
22         x = int(sys.argv[1])

```

```

22     y = int(sys.argv[2])
23     else:
24         print usage()
25         sys.exit(1)
26     print "Requesting %s+%s"%(x, y)
27     print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))

```

Agora em C++:

Na linha 2 *beginner_tutorials/AddTwoInts.h* é o arquivo *header* gerado pelo arquivo *.srv* de exemplos anteriores.

A função *add* provê o serviço para adicionar dois inteiros, ela recebe uma requisição e responde com o tipo definido no arquivo *.srv*, além de retornar um booleano. Nas linhas de 5 a 10 os dois inteiros são adicionados e guardados na resposta (Response), então algumas informações sobre a requisição (request) são guardadas. Por último, o serviço retorna verdadeiro quando é completado.

Na linha 17 o serviço é passado pelo ROS, com o nome de "*add_two_ints*", e seu *handle* é a função *add*.

add_two_ints_server.cpp

```

1  #include "ros/ros.h"
2  #include "beginner_tutorials/AddTwoInts.h"
3
4  bool add(beginner_tutorials::AddTwoInts::Request &req,
5           beginner_tutorials::AddTwoInts::Response &res)
6  {
7      res.sum = req.a + req.b;
8      ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (
9          long int)req.b);
10     ROS_INFO("sending back response: [%ld]", (long int)
11         res.sum);
12     return true;
13 }
14
15 int main(int argc, char **argv)
16 {
17     ros::init(argc, argv, "add_two_ints_server");
18     ros::NodeHandle n;
19
20     ros::ServiceServer service = n.advertiseService("
21         add_two_ints", add);
22     ROS_INFO("Ready to add two ints.");
23     ros::spin();
24
25     return 0;
26 }

```

Agora o código do cliente:

add_two_ints_client.cpp

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)
6 {
7     ros::init(argc, argv, "add_two_ints_client");
8     if (argc != 3)
9     {
10         ROS_INFO("usage: add_two_ints_client X Y");
11         return 1;
12     }
13
14     ros::NodeHandle n;
15     ros::ServiceClient client = n.serviceClient<
16         beginner_tutorials::AddTwoInts>("add_two_ints");
17     beginner_tutorials::AddTwoInts srv;
18     srv.request.a = atoll(argv[1]);
19     srv.request.b = atoll(argv[2]);
20     if (client.call(srv))
21     {
22         ROS_INFO("Sum: %ld", (long int) srv.response.sum);
23     }
24     else
25     {
26         ROS_ERROR("Failed to call service add_two_ints");
27         return 1;
28     }
29     return 0;
30 }
```

Na linha 15 o cliente é criado para o serviço `add_two_ints`. O objeto `ros::ServiceClient` é usado para chamar o serviço posteriormente. Na linha 16 uma classe de serviço autogerada é instanciada e nas linhas 17 e 18 os valores para o membro de requisição (*request*) são assinalados. Qualquer classe de serviço (*service class*) contém dois membros: *request* e *response*, também contém duas definições de classe: *Response* e *Request*.

Na linha 19 é onde de fato o serviço é chamado. Como as chamadas (*calls*) do serviço são bloqueadas, ele retornará uma única vez quando a chamada for feita. Se a chamada do serviço for bem sucedida, a função *call()* retornará verdadeiro e o valor em *srv.response* será válido. Se a chamada não for bem sucedida, *call()* retornará falso e o valor dentro da resposta de *srv.response* será inválido.

Com todos os códigos em mãos é possível fazer vários testes no terminal. Basta lembrar

de rodar o nó de serviço antes de chamar o cliente.

6.2 Parâmetros

O *rosparam* é uma maneira fácil de estabelecer parâmetros globais a todos os nós, que podem ser acessados ou modificados por eles. Possibilitando assim a troca de *flags* entre os programas, indicações de estados, indicações de início ou finalização de tarefas, dentre outras coisas.

Os parâmetros podem ser manipulados no terminal pelo comando *rosparam*. Por exemplo, para se definir um pode-se usar:

```
$ rosparam set <nome> <valor>
```

Para ter acesso ao valor de um parâmetro:

```
$ rosparam get <nome>
```

Para listar todos eles:

```
$ rosparam list
```

Dentre outras possibilidades.

Desta forma, dentro de um nó em python, por exemplo, um parâmetro pode ser acessado e modificado com:

```
1 variavel = rospy.get_param('/parametro')
2
3 rospy.set_param('/parametro', valor)
```

Em C++:

```
1 ros::param::get("/parametro", variavel);
2
3 ros::param::set("/parametro", valor);
```

Parâmetros são, usualmente, declarados em arquivos launch, que serão abordados um pouco mais a frente neste capítulo.

6.3 Servidor de Reconfiguração Dinâmica

Agora evoluir o conceito de parâmetros será aprofundado e será abordado como utilizar um servidor dinâmico de parâmetros no ROS. Esse tipo de ferramenta é útil quando diferentes nós utilizam e precisam fazer alterações no valor de um mesmo parâmetro do sistema robótico.

OBS : É possível programar um nó dinamicamente reconfigurável em C++, mas nesta apostila será trabalhado apenas com exemplos em Python.

6.3.1 Arquivo .cfg

Um diretório chamado `cfg` será criado dentro do pacote:

```
$ cd ~/catkin_ws/src/beginner_tutorials
$ mkdir cfg
```

Agora, um arquivo chamado `Tutorials.cfg` que deve ter a seguinte estrutura:

Tutorials.cfg

```
1 #!/usr/bin/env python
2 PACKAGE = "beginner_tutorials"
3
4 from dynamic_reconfigure.parameter_generator_catkin import
5     *
6 gen = ParameterGenerator()
7
8 gen.add("int_param",    int_t,    0, "An Integer parameter"
9     , 50, 0, 100)
10 gen.add("double_param", double_t, 0, "A double parameter",
11     .5, 0, 1)
12 gen.add("str_param",    str_t,    0, "A string parameter",
13     "Hello World")
14 gen.add("bool_param",   bool_t,   0, "A Boolean parameter",
15     True)
16
17 gen.add("W_param",     int_t,     0, "how many times??", 0, 0,
18     10000)
19
20 size_enum = gen.enum([ gen.const("Small",      int_t, 0, "A
21     small constant"),
22     gen.const("Medium",      int_t, 1, "A
23     medium constant"),
24     gen.const("Large",      int_t, 2, "A
25     large constant"),
26     gen.const("ExtraLarge", int_t, 3, "
27     An extra large constant")],
28     "An enum to set size")
29
30 gen.add("size", int_t, 0, "A size parameter which is edited
31     via an enum", 1, 0, 3, edit_method=size_enum)
32
33 exit(gen.generate(PACKAGE, "dynamic_tutorials", "Tutorials"
34     ))
```

As linhas de 1 a 4 servem para inicializar o ROS e importar o gerador de parâmetros, enquanto que na linha 6 esse gerador é definido.

Tendo um gerador já é possível começar a definir uma lista de parâmetros. Para fa-

zer isso, alguns argumentos serão necessários:

- nome: uma string que especifica o nome sob o qual esse parâmetro deve ser armazenado.
- tipo de parâmetro: define o tipo de valor armazenado e pode ser qualquer um de `int_t`, `double_t`, `str_t` ou `bool_t`
- level: uma bitmask que será passada posteriormente para o retorno de chamada de reconfiguração dinâmica. Quando o retorno de chamada é feito, é feita uma operação OR com todos os valores de level (ou nível) dos parâmetros alterados e o valor resultante é passado para o retorno de chamada.
- descrição: string que descreve o parâmetro.
- default: especifica o valor padrão do parâmetro.
- min: especifica o valor mínimo (opcional e não se aplica a strings e booleans).
- max: especifica o valor máximo (opcional e não se aplica a strings e booleans).

As linhas de 8 a 11 exemplificam como é feita a definição de parâmetros. Já as linhas 13 a 19 definem um número inteiro cujo valor é definido por uma enumeração, e para fazer isso, é chamado *gen.enum* e passado uma lista de constantes seguida por uma descrição do *enum*. Após ter criado um *enum*, pode-se transmiti-lo ao gerador, e então o parâmetro pode ser definido com "*Small*" ou "*Medium*" em vez de 0 ou 1.

A última linha diz ao gerador para gerar os arquivos necessários e sair do programa. O segundo argumento é o nome de um nó no qual ele pode ser executado (usado apenas para gerar documentação), o terceiro argumento é um prefixo de nome que os arquivos gerados vão receber (por exemplo, "*<name>Config.h*" para C++, ou "*<name>Config.py*" para Python).

Para utilizar o arquivo *.cfg* primeiro precisa-se torná-lo executável:

```
$ chmod a+x cfg/Tutorials.cfg
```

Em seguida, é preciso adicionar as seguintes linhas ao *CMakeLists.txt*:

```
#add dynamic reconfigure api
#find_package(catkin REQUIRED dynamic_reconfigure)
generate_dynamic_reconfigure_options(
    cfg/Tutorials.cfg
    #...
)
```

```
# make sure configure headers are built before any node using them
add_dependencies(example_node ${PROJECT_NAME}_gencfg)
```

Depois de compilar o *workspace* já vai ser possível utilizar o *Tutorials.cfg*.

6.3.2 Configurando o Nó Servidor Dinâmico

Depois de criado um arquivo *.cfg*, será abordado como programar um nó dinamicamente reconfigurável. Dentro do diretório *scripts* deve-se criar um arquivo chamado *dyn_reconfigure_server.py*.

dyn_reconfigure_server.py

```

1 #!/usr/bin/env python
2
3 import rospy
4
5 from dynamic_reconfigure.server import Server
6 from beginner_tutorials.cfg import TutorialConfig
7
8 def callback(config, level):
9     rospy.loginfo(""" Reconfigure Request: {} """.format(
10         config))
11     active config
12
13 if __name__ == "__main__":
14     rospy.init_node("Tutorial", anonymous = False)
15
16     srv = Server(TutorialConfig, callback)
17     rospy.spin()

```

Nas linhas 3 e 5 as dependências necessárias do ROS e da reconfiguração dinâmica são importadas, mas além disso, na linha 6 o tipo de configuração presente no diretório `cfg` é importado. O nome `TutorialConfig` é gerado automaticamente anexando `Config` ao terceiro argumento em `gen.generate`, como explicado na parte anterior.

As linhas de 8 a 10 contém a definição da função `callback` que será chamada quando a configuração for atualizada. Neste caso ela irá imprimir a lista de parâmetros com a configuração atualizada. No entanto, a `callback` também poderia editar a configuração antes de retorná-la na linha 10. Por exemplo, se um usuário definir parâmetros conflitantes, a `callback` deve configurá-los para um estado válido.

Por último, nas linhas finais do código o nó é inicializado e um servidor construído, passando o tipo de configuração e a função de retorno de chamada como argumento.

Para utilizar o nó em Python antes temos de torná-lo executável:

```
$ chmod +x scripts/dyn_reconfigure_server.py
```

6.3.3 Exemplo de Cliente

Este primeiro exemplo de utilização do servidor dinamicamente será simples. Será programado um cliente que irá atualizar a configuração a cada dez segundos, e então o cliente e o servidor começarão a imprimir as configurações correspondentes.

Segue o código do cliente:

dyn_client.py

```

1 #!/usr/bin/env python
2
3 import rospy
4

```

```

5 import dynamic_reconfigure.client
6
7 def callback(config):
8     rospy.loginfo("Config set to {int_param}, {double_param
9         }, {str_param}, {bool_param}, {size}".format(**
10         config))
11
12
13 if __name__ == "__main__":
14     rospy.init_node("dynamic_client")
15
16     client = dynamic_reconfigure.client.Client("Tutorial",
17         timeout=30, config_callback=callback)
18
19     r = rospy.Rate(0.1)
20     x = 0
21     b = False
22     while not rospy.is_shutdown():
23         x = x+1
24         if x>10:
25             x=0
26         b = not b
27         client.update_configuration({"int_param":x, "
28             double_param":(1/(x+1)), "str_param":str(rospy.
29             get_rostime()), "bool_param":b, "size":1})
30         r.sleep()

```

Primeiro, na linha 5 o *dynamic_reconfigure.client* é importado. Em seguida, nas linhas 7 e 8 define-se um retorno de chamada que imprimirá a configuração retornada pelo servidor. Há duas diferenças principais entre esse retorno de chamada e o do servidor, uma é que esta *callback* não precisa retornar um objeto de configuração atualizado, e a outra é que não possui o argumento "level". Esse retorno de chamada é opcional.

Por fim, inicializa-se o ROS na linha 11 e o cliente na linha 13. O loop principal é executado uma vez a cada dez segundos e chama *update_configuration* no cliente após ter alterado os valores dos parâmetros. Observa-se que não é preciso usar uma configuração completa e também pode-se passar um dicionário com apenas um dos parâmetros.

Inicia-se o *roscore*, o nó do servidor e, em seguida, o novo nó do cliente.

6.3.4 Exemplo de Cliente e Serviço de Tipo Trigger

Neste segundo exemplo de cliente do servidor dinâmico de parâmetros será mostrada uma aplicação de serviço *Trigger*. Esse é um tipo de serviço padrão do ROS bastante útil, pois sua requisição (*request*) não possui nenhum campo. Segue abaixo a definição do *Trigger.srv* para facilitar o entendimento.


```
bool success
string message
```

Como é observado, não há nada definido antes da linha com '—', o que significa que o serviço só tem campos na resposta (*response*). O primeiro campo é um booleano que indica se o serviço funcionou da maneira correta, e o segundo campo pode conter informações sobre a execução do serviço, como por exemplo mensagens de erro.

O seguinte cenário será suposto para exemplificar esta aplicação: é necessário fazer a computação de algumas variáveis do sistema, mas o *script* que analisa se essa computação deve ser feita não pode se ocupar com essa tarefa, por exemplo, o robô é controlado através de uma interface gráfica e o valor das variáveis são controlados por cliques nos botões dessa interface. Caso a interface pare de checar os cliques para alterar algumas variáveis do servidor dinâmico, ou ela perderia algumas requisições de alteração nos valores, ou no pior dos casos, poderia até mesmo *crashar*. Em um cenário desse tipo, rodar um serviço *Trigger* é uma excelente alternativa.

trigger_service.py

```
1 #!/usr/bin/env python
2
3 import rospy
4 import dynamic_reconfigure.client
5
6 from std_srvs.srv import Trigger, TriggerResponse
7 import time
8
9 def handle_service(req):
10     w = rospy.get_param('/Tutorials/w_param')
11     w += 1
12     while w:
13         c = 0
14         c += 1
15         if c == 5:
16             dyn_reconfig_client.update_configuration({'
17                 w_param': w})
18             return TriggerResponse(success=True, message="
19                 we have waited too long already")
20         time.sleep(1)
21     continue
22
23 def start_service():
24     rospy.init_node('service_node')
25     s = rospy.Service('trigger_service', Trigger,
26                       handle_service)
27     rospy.loginfo("trigger service running")
28     rospy.spin()
29
30 def emit(config):
```

```

28     rospy.loginfo('emitting')
29
30 if __name__ == "__main__":
31     try:
32         dyn_reconfig_client = dynamic_reconfigure.client.
            Client("Tutorial", timeout=3, config_callback=
                emit)
33     except rospy.ROSException as e:
34         rospy.logwarn("dynamic reconfigure not found!!")
35         break
36
37     start_service()

```

Na linha 10 carrega-se o parâmetro do servidor com o método *get_param()*. Após cinco segundos, o serviço vai terminar sua tarefa e retornar a *Response* carregando o booleano e a mensagem, linha 17.

Neste exemplo, também é usada uma construção *try/except*, linhas 31 a 35, que checa se o servidor de parâmetros está rodando ou não, e caso não esteja, o serviço não é iniciado e uma mensagem de alerta é mostrada no terminal. Caso tudo esteja rodando e o serviço seja iniciado, a linha 24 faz com que uma outra mensagem seja exibida. Além disso, nesse código a função *callback* do cliente apenas imprime uma mensagem de texto quando a configuração do servidor é atualizada.

Abaixo um exemplo de cliente que chama esse serviço.

trigger_client.py

```

1  #!/usr/bin/env python
2
3  import rospy
4  import dynamic_reconfigure.client
5
6  from std_srvs.srv import Trigger, TriggerRequest
7
8  def client(req):
9      try:
10         rospy.wait_for_service('trigger_service', 2.0)
11         s = rospy.ServiceProxy('/trigger_service', Trigger)
12         t_req = TriggerRequest()
13         result = s(t_req)
14     except rospy.ServiceException, e:
15         print "Service call failed: %s"%e
16     except Exception as e:
17         rospy.logwarn("Service not found!!")
18         print "Service call failed with generic Exception:
            %s"%e
19
20 def emit(config):
21     rospy.loginfo('emitting')

```

```

22
23 if __name__ == "__main__":
24     try:
25         dyn_reconfig_client = dynamic_reconfigure.client.
                Client("Tutorial", timeout=3, config_callback=
                emit)
26     except rospy.ROSException as e:
27         rospy.logwarn("dynamic reconfigure not found!!")
28         break
29
30     client()

```

Na linha 10 tenta-se conectar com o serviço, caso não funcione, a mensagem da linha 15 é impressa. Em seguida, na linha 11 a variável *s* recebe o *handle* do serviço *trigger_service* que é do tipo *Trigger*. Caso falhe, serão exibidas as mensagens nas linhas 17 e 18. Na linha 12 cria-se um objeto *TriggerRequest*, e na linha 13 envia-se a requisição através de *s*.

6.4 Remap e Namespace

Nomeação de nós e tópicos é crucial para o funcionamento do ROS. Nós com o mesmo nome não podem ser executados. Por isso, inclusive, pode ser definido o argumento *anonymous=True* naqueles feitos em Python.

No entanto, os nomes dos nós e dos tópicos podem ser alterados para evitar conflitos. Este processo é conhecido como **remap**.

Pode-se ainda alterar a estrutura hierárquica dos nomes, criando um **namespace**, ou seja, um bloco no qual eles estarão subordinados.

Para se fazer um remap no terminal ao se iniciar o nó basta adicionar parâmetros. Para mudar o nome do tópico adiciona-se *<nome>:=<novo nome>* e para alterar o nome do próprio nó *__name:<novo nome>*. O exemplo a seguir utiliza o nó *talker.py* da página 36 para fazer um remap do tópico *chatter* para *chat1* e do próprio nome do nó para *t1*:

```

$ rosrn beginner_tutorials talker.py chatter:=chat1
  __name:=t1
# Novo terminal
$ rostopic list
/chat1
/rosout
/rosout_agg
$ rosnode list
/rosout
/t1

```

Sendo assim, é possível executar dois nós do tipo *talker* e dois nós do tipo *listener* como a Fig. 4.3.1 da página 44, porém sem causar interferência entre eles:

```

# Em terminais diferentes
$ rosrn beginner_tutorials talker chatter:=chat1 __name
:=talker1

```

```

$ rosrn beginner_tutorials listener chatter:=chat1
  __name:=listener1
$ rosrn beginner_tutorials talker chatter:=chat2 __name
:=talker2
$ rosrn beginner_tutorials listener chatter:=chat2
  __name:=listener2

```

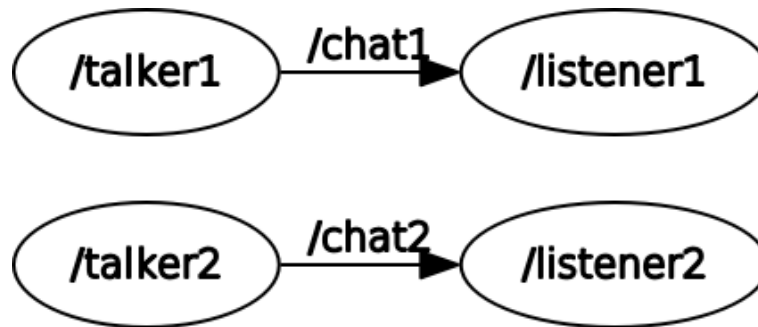


Figura 6.4.1: Grafo dois talker e dois listener sem interferência

6.5 Launch

Em sistemas mais complexos, há vários programas parem serem executados, e fazê-lo um a um em vários terminais pode ser chato e cansativo. Para isso, é possível condensar as inicializações em um único arquivo, este arquivo irá abrir nó por nó, além de poder realizar facilmente *remaps*, *namespaces*, definir parâmetros, dentre outras funcionalidades. Esse arquivo é chamado de **launch**. Geralmente, é salvo em uma pasta também chamada de *launch*, dentro do pacote.

Todo o conteúdo do arquivo fica contido entre `<launch>` e `</launch>`. Cada nó é definido entre `<node>` e `</node>`, e assim por diante. Um exemplo que executa o `turtlesim` e o `teleop` é mostrado a seguir:

turtleteleop.launch

```

1 <launch >
2
3   <node pkg=" turtlesim " name=" sim " type=" turtlesim_node ">
4     </node >
5
6   <node pkg=" turtlesim " name=" teleop " type="
7     turtle_teleop_key ">
8     </node >
9 </launch >

```

Para se definir um parâmetro em um arquivo de launch, deve-se incluir a instrução na linha 2:

```

1 <launch >

```

```

2     <param name="nome" type="tipo" value="valor" />
3 </launch>

```

Na qual as aspas devem ser mantidas.

O exemplo a seguir realiza dois grupos de namespace para descrever dois nós *turtle_sim* diferentes, e também faz um remap de tópicos para que o nó *mimic* opere corretamente. O resultado final será a tartaruga *turtlesim2* imitará os movimentos da tartaruga *turtlesim1*.

mimic.launch

```

1 <launch>
2
3     <group ns="turtlesim1">
4         <node pkg="turtlesim" name="sim" type="
5             turtlesim_node"/>
6     </group>
7
8     <group ns="turtlesim2">
9         <node pkg="turtlesim" name="sim" type="
10            turtlesim_node"/>
11    </group>
12
13    <node pkg="turtlesim" name="mimic" type="mimic">
14        <remap from="input" to="turtlesim1/turtle1"/>
15        <remap from="output" to="turtlesim2/turtle1"/>
16    </node>
17 </launch>

```

Também é possível carregar um arquivo de parâmetros no formato *.yaml* antes de iniciar um nó. Ter todos os parâmetros do pacote definidos em um único arquivo ajuda a facilitar a sua utilização em diferentes sistemas. Para isso, dentro de um diretório chamado *config* serão salvos os arquivos *.yaml*, um exemplo da estrutura desse tipo de arquivo pode ser conferido abaixo.

```

# Radius of the wheels (meters)
R: 0.1997
# Distance between the center wheels (meters)
L: 0.4347
# Timeout, how long after stop receiving commands to send 0 velocity to
TIME_OUT: 0.5

```

Antes da definição de cada um dos parâmetros há um comentário com uma breve descrição sobre ele. Para definir um parâmetro basta colocar o nome, em seguida dois pontos, e então o valor. O exemplo de launch a seguir mostra como importar esse arquivo de parâmetros.

params.launch

```

1 <launch >
2
3   <node pkg="beginner_tutorials" name="<nome_do_no>" type
4     = "<arquivo_onde_o_no_foi_definido>" args="" output="
5       screen">
6   <roscparam command="load" file="$(find
7     beginner_tutorials)/config/<nome>.yaml"/>
8   </node>
9 </launch >

```

Para iniciar um servidor dinâmico de parâmetros, ou disponibilizar um serviço, basta fazer como no exemplo abaixo.

dyn_service.launch

```

1 <launch >
2
3   <node pkg="beginner_tutorials" name="dyn_server" type="
4     dyn_reconfigure_server.py" respawn="true" />
5   <node pkg="beginner_tutorials" name="trigger_service"
6     type="trigger_service.py" args="" output="screen" />
7 </launch >

```

6.6 Múltiplas Máquinas

O ROS pode ser usado através de múltiplas máquinas, desde que estejam conectadas através de uma mesma rede. O procedimento é simples, visto que ele também foi desenvolvido para tarefas como esta.

Neste caso, apenas uma máquina será o *master*, ou seja, apensar uma máquina executará o comando *roscore*, e todos os nós deverão estar configurados para esse *master*, pelo comando *ROS_MASTER_URI*.

O exemplo a seguir usará uma máquina chamada *alfa* que possui um IP genérico XXX.XXX.X.XX como *master* e outra chamada *beta* com IP YYY.YYY.Y.YY.

DICA: Para adquirir o endereço IP de uma máquina basta usar o comando no terminal:

```
$ hostname -I
```

Sendo assim, em alfa, deve-se usar os comandos:

```
$ export ROS_IP=XXX.XXX.X.XX
$ roscore
```

Agora, na máquina beta, os comandos:

```
$ export ROS_MASTER_URI=http://XXX.XXX.X.XX:11311
$ export ROS_IP=YYY.YYY.Y.YY
```

No exemplo, está sendo usada a porta padrão do ROS, 11311. Nota-se também, que não foi executado o comando *roscore* em Beta.

A conexão já está estabelecida. Para o teste será usado o *turtlesim*, visto na seção **Nós e Tópicos**, na página 27.

Em beta:

```
#EM BETA
$ rosrunc turtlesim turtlesim_node
```

É possível controlar a tartaruga de beta por alfa:

```
#EM ALFA
$ rosrunc turtlesim turtle_teleop_key
```

Para tornar os procedimentos automáticos, os comandos executados em cada máquina para estabelecer a conexão podem ser adicionados ao final do arquivo *.bashrc*. Isso pode ser útil para executar simulações em que, por exemplo, há uma pequena máquina acoplada a um robô móvel, responsável por controlá-lo que necessite de comandos vindos de outra máquina.

Problemas comuns

É recorrente que, ao configurar as máquinas para comunicação, os tópicos são listados (através de *rostopic list*) mas as mensagens dos tópicos não são transmitidas (nada se obtém com *rostopic echo*).

Este problema pode ser causado por dois fatores. O primeiro se deve ao fato das máquinas não conhecerem, dentro da rede, os nomes uma da outra.

Para solucionar, abra o arquivo */etc/hosts* em alpha e insira a linha *YYY.YYY.Y.YYY beta*. Faça o equivalente em beta, inserindo a linha *XXX.XXX.X.XXX alpha* no arquivo */etc/hosts*. Como exemplo, o arquivo */etc/hosts* vai ficar parecido com o exemplo abaixo:

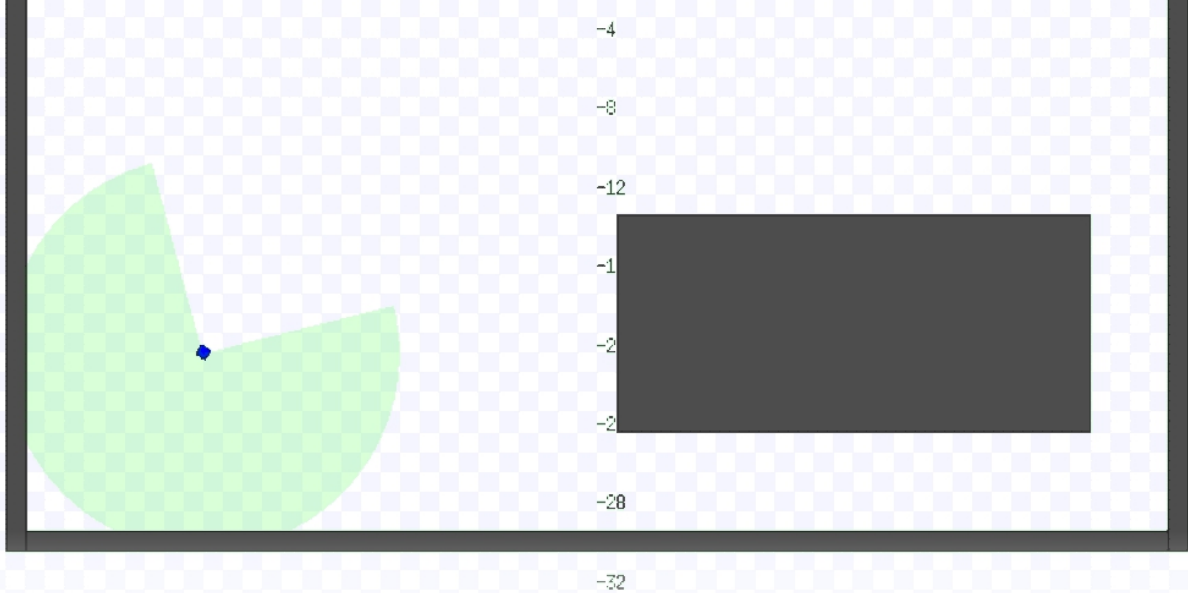
```
127.0.0.1      localhost
127.0.1.1      alpha
YYY.YYY.Y.YYY beta
```

O segundo problema está relacionado a um bloqueio do firewall. Para solucionar, insira o seguinte comando em ambas as máquinas:

```
#EM ALFA E EM BETA
$ sudo ufw disable
```


Simulações

7	Stage	75
8	RViz	83
9	Gazebo	97
10	CoppeliaSim	107
10.1	Introdução	
10.2	Instalação	
10.3	Interface Gráfica do Usuário	
10.4	Comunicação Remota	
11	Teleop ROSI	113
11.1	Baixo Nível	
11.2	Launch	
12	Cenário de um Manipulador	119
12.1	Cenário	
12.2	Controle Externo	
13	Controle de Quadrotor	127
13.1	Teoria	
13.2	Cenário	
13.3	Implementação	



7. Stage

Este capítulo irá abordar o simulador Stage através de exemplos de usos. Junto com o Gazebo, eles representam os principais simuladores do ROS. O Stage é mais enxuto e comporta sistemas bidimensionais. Já o segundo, Gazebo, é mais completo e pode demandar um pouco mais de esforço da CPU e GPU do computador. Além disso, há ainda uma ferramenta de visualização muito potente, o RViz. Com ela é possível acompanhar a simulação visualizando mapa, marcadores, eixos, frames, etc.

É possível executar o simulador da seguinte forma:

```
$ rosrund stage_ros stageros -d
```

Assim, abrirá uma nova janela com o simulador. Pressionando R uma vez, é possível colocar o mapa em perspectiva. O argumento *-d* servirá para indicar o arquivo mapa a ser aberto.

Um novo pacote será criado para conter os arquivos utilizados nesta simulação:

```
# Na workspace
$ catkin_create_pkg example rospy roscpp
$ roscd example
$ mkdir launch
$ mkdir scripts
$ mkdir src
$ mkdir worlds
```

Dentro de *worlds*, ficarão as configurações para o mundo do simulador, necessariamente um arquivo *png* e um arquivo *world*. O primeiro deles será uma imagem de 60px x 60px, contendo apenas pixels brancos e pretos como a Fig. 7.0.1.

Feito isso, o arquivo *world* contém o seguinte código:

map_1.world

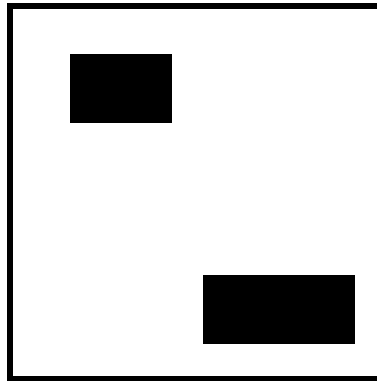


Figura 7.0.1: Arquivo *map_1.png*

```

1  define block model
2  (
3    size [0.500 0.500 0.500]
4    gui_nose 0
5  )
6
7  define topurg ranger
8  (
9    sensor(
10   range [ 0.0  10.0 ]
11   fov 270.25
12   samples 270
13  )
14
15  # generic model properties
16  color "black"
17  size [ 0.050 0.050 0.100 ]
18  )
19
20  define erratic position
21  (
22
23   size [0.550 0.550 0.250]
24
25   origin [-0.050 0.000 0.000 0.000]
26   gui_nose 1
27   drive "diff"
28   topurg(pose [ 0.050 0.000 0.000 0.000 ])
29  )
30
31
32
33  define floorplan model
34  (
35   # sombre , sensible , artistic

```

```
36     color "gray30"
37
38     # most maps will need a bounding box
39     boundary 1
40
41     gui_nose 0
42     gui_grid 0
43
44     gui_outline 0
45     gripper_return 0
46     fiducial_return 0
47     laser_return 1
48 )
49
50 # set the resolution of the underlying raytrace model in
51   meters
52 resolution 0.02
53
54 interval_sim 50 # simulation timestep in milliseconds
55
56 window
57 (
58     size [ 700 700 ]
59
60     rotate [ 0.000 0.000 ]
61     scale 10.000
62
63     # GUI options
64     show_data 1
65     show_blocks 1
66     show_flags 1
67     show_clock 1
68     show_follow 0
69     show_footprints 1
70     show_grid 1
71     show_status 1
72     show_trailarrows 0
73     show_trailrise 0
74     show_trailfast 0
75     show_occupancy 0
76     show_tree 0
77     pcam_on 0
78     screenshots 0
79 )
80
81 # load an environment bitmap
82 floorplan
```

```

83 (
84   name "my_map"
85   bitmap "map_1.png"
86   size [60.000 60.000 0.500]
87   pose [ 0.000 0.000 0.000 0.000 ] #[x y ? theta]
88 )
89
90 # throw in a robot
91 erratic( pose [ -20.000 -20.000 0.000 -120.000 ] name "
    my_robot" color "blue")
92 #block( pose [ -13.000 18.000 0.000 180.000 ] name "my_goal"
    " color "red")

```

O código define o robô a ser simulado e o mapa. Feito isso, pode-se executar o Stage da seguinte forma:

```

$ rosrund stage_ros stageros -d ./src/example/worlds/
  map_1.world

```

Abrirá uma janela como a Fig. 7.0.2.

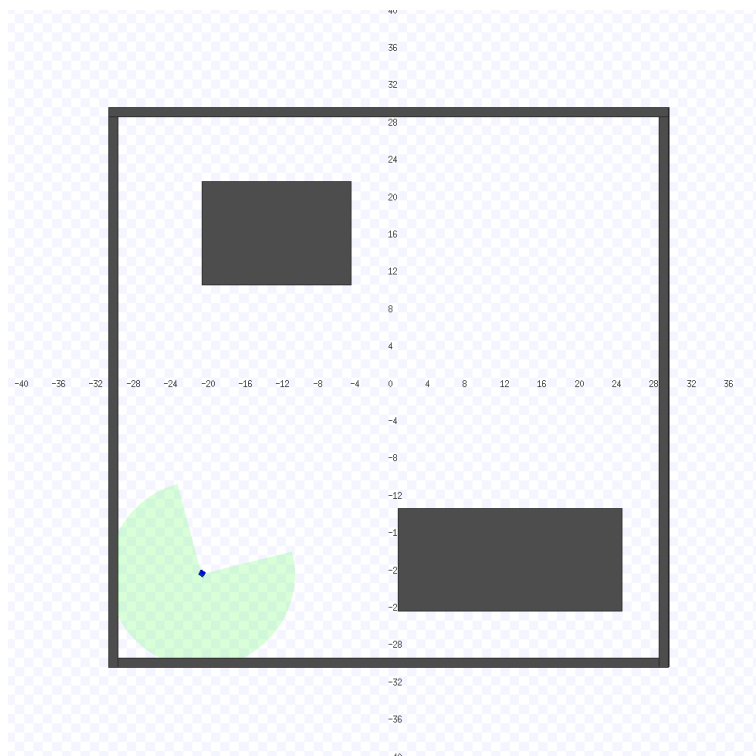


Figura 7.0.2: Execução do Stage

Por fim, basta controlar o robô com um nó como os criados anteriormente para controlar a tartaruga *turtlesim*.

example_node.py

```

1 #!/usr/bin/env python

```

```
2 import rospy
3 from geometry_msgs.msg import Twist
4 from nav_msgs.msg import Odometry
5 from math import sqrt, atan2, exp, atan, cos, sin, acos, pi
  , asin, atan2
6 from tf.transformations import euler_from_quaternion,
  quaternion_from_euler
7 from time import sleep
8 from visualization_msgs.msg import Marker, MarkerArray
9 import tf
10 import sys
11 # Frequencia de simulacao no stage
12 global freq
13 freq = 20.0 # Hz
14 # Velocidade de saturacao
15 global Usat
16 Usat = 5
17 # Estados do robo
18 global x_n, y_n, theta_n
19 x_n = 0.1 # posicao x atual do robo
20 y_n = 0.2 # posicao y atual do robo
21 theta_n = 0.001 # orientacao atual do robo
22 # Estados do robo
23 global x_goal, y_goal
24 x_goal = 0
25 y_goal = 0
26 # Relativo ao feedback linearization
27 global d
28 d = 0.80
29 # Relativo ao controlador (feedforward + ganho proporcional
  )
30 global Kp
31 Kp = 1
32 # Rotina callback para a obtencao da pose do robo
33 def callback_pose(data):
34     global x_n, y_n, theta_n
35     x_n = data.pose.pose.position.x # posicao 'x' do robo
  no mundo
36     y_n = data.pose.pose.position.y # posicao 'y' do robo
  no mundo
37     x_q = data.pose.pose.orientation.x
38     y_q = data.pose.pose.orientation.y
39     z_q = data.pose.pose.orientation.z
40     w_q = data.pose.pose.orientation.w
41     euler = euler_from_quaternion([x_q, y_q, z_q, w_q])
42     theta_n = euler[2] # orientaco 'theta' do robo no
  mundo
43     return
```

```

44 # Rotina para a geracao da trajetoria de referencia
45 def refference_trajectory(time):
46     ##MUDAR PARA CIRCULO
47     global x_goal, y_goal
48     x_ref = x_goal
49     y_ref = y_goal
50     Vx_ref = 0
51     Vy_ref = 0
52     return (x_ref, y_ref, Vx_ref, Vy_ref)
53 # Rotina para a geracao da entrada de controle
54 def trajectory_controller(x_ref, y_ref, Vx_ref, Vy_ref):
55     global x_n, y_n, theta_n
56     global Kp
57     global Usat
58     Ux = Vx_ref + Kp * (x_ref - x_n)
59     Uy = Vy_ref + Kp * (y_ref - y_n)
60     absU = sqrt(Ux ** 2 + Uy ** 2)
61     if (absU > Usat):
62         Ux = Usat * Ux / absU
63         Uy = Usat * Uy / absU
64     return (Ux, Uy)
65 # Rotina feedback linearization
66 def feedback_linearization(Ux, Uy):
67     global x_n, y_n, theta_n
68     global d
69     VX = cos(theta_n) * Ux + sin(theta_n) * Uy
70     WZ = (-sin(theta_n) / d) * Ux + (cos(theta_n) / d) * Uy
71     return (VX, WZ)
72 # Rotina primaria
73 def example():
74     global freq
75     global x_n, y_n, theta_n
76     global pub_rviz_ref, pub_rviz_pose
77     vel = Twist()
78     i = 0
79     pub_stage = rospy.Publisher("/cmd_vel", Twist,
80                                queue_size=1) #declaracao do topico para comando de
81                                velocidade
82     rospy.init_node("example_node") #inicializa o no "este
83     no"
84     rospy.Subscriber("/base_pose_ground_truth", Odometry,
85                      callback_pose) #declaracao do topico onde sera lido
86     o estado do robo
87     #Inicializa os nos para enviar os marcadores para o
88     rviz
89     pub_rviz_ref = rospy.Publisher("/
90     visualization_marker_ref", Marker, queue_size=1) #
91     rviz marcador de velocidade de referencia

```



```

84     pub_rviz_pose = rospy.Publisher("/
        visualization_marker_pose", Marker, queue_size=1) #
        rviz marcador de velocidade do robo
85     # Define uma variavel que controlar[a a frequencia de
        execucao deste no
86     rate = rospy.Rate(freq)
87     sleep(0.2)
88     # O programa do no consiste no codigo dentro deste
        while
89     while not rospy.is_shutdown(): #" Enquanto o programa
        nao ser assassinado "
90         # Incrementa o tempo
91         i = i + 1
92         time = i / float(freq)
93         # Obtem a trajetoria de referencia " constante neste
        exemplo "
94         [x_ref, y_ref, Vx_ref, Vy_ref] =
            refference_trajectory(time)
95         # Aplica o controlador
96         [Ux, Uy] = trajectory_controller(x_ref, y_ref,
            Vx_ref, Vy_ref)
97         # Aplica o feedback linearization
98         [V_forward, w_z] = feedback_linearization(Ux, Uy)
99         # Publica as velocidades
100        vel.linear.x = V_forward
101        vel.angular.z = w_z
102        pub_stage.publish(vel)
103        # Espera por um tempo de forma a manter a frequencia
        desejada
104        rate.sleep()
105 # Funcao inicial
106 if __name__ == '__main__':
107     # Obtem os argumentos , no caso a posicao do alvo
108     x_goal = int(sys.argv[1])
109     y_goal = int(sys.argv[2])
110     try:
111         example()
112     except rospy.ROSInterruptException:
113         pass

```

Executando o nó em um novo terminal, pode-se observar a convergência do robô para o ponto desejado por feedback linearization, 48. No caso exemplo seguir, o ponto alvo é (0,0).

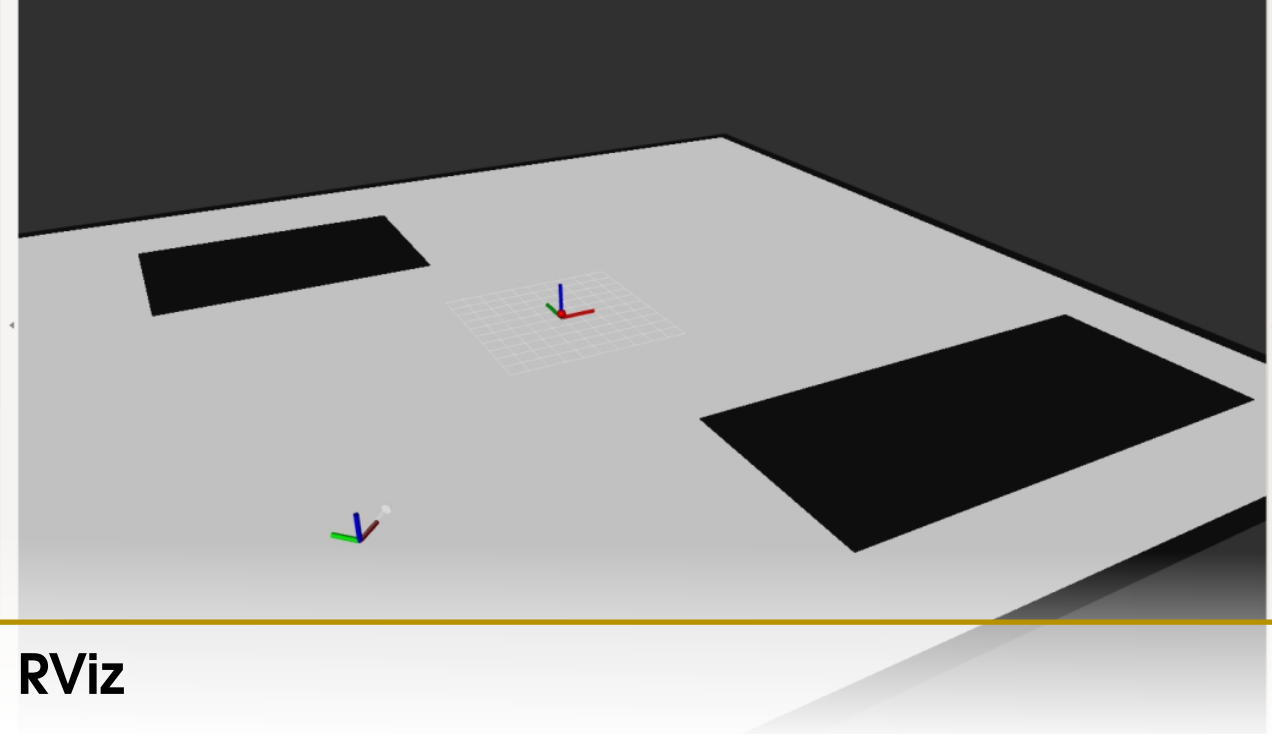
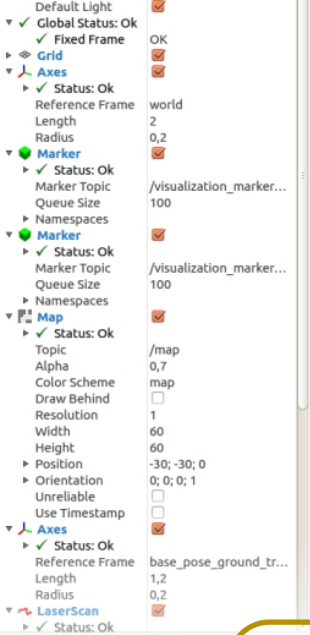
```
$ rosrund example example_node.py 0 0
```

Os nós podem ser colocados em um único arquivo *launch*:

example.launch

```
1 <?xml version="1.0"?>
```

```
2
3 <launch>
4
5 <!--Run the stage simulator-->
6 <node pkg = "stage_ros" name = "stageros" type = "stageros"
   output = "screen" args="-d $(find example)/worlds/map_1
   .world">
7 </node>
8
9 <!--Run the controller node      args="x_goal y_goal" -->
10 <node pkg = "example" name = "example_node" type = "
   example_node.py" args="0 0" output="screen">
11 </node>
12
13 </launch>
```



8. RViz

Este capítulo usará a simulação no Stage do capítulo anterior como modelo para o uso e explicação do RViz. É necessário instalar um pacote para visualização do mapa, caso ainda não tenha sido feito.

```
$ sudo apt-get install ros-kinetic-map-server
```

Criando novas pastas para guardar os arquivos de configuração do RViz e do mapa:

```
$ roscd example
$ mkdir maps
$ mkdir rviz
```

Dentro de maps ficará a mesma imagem da Fig. 7.0.1 e um arquivo *yaml* preenchido com:

map_1.yaml

```
image: map_1.png
resolution: 1
origin: [-30.0, -30.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

Além disso, dentro da pasta rviz deve existir um arquivo com as configurações iniciais do visualizador:

rviz_config.rviz

```
Panels:
- Class: rviz/Displays
  Help Height: 78
  Name: Displays
```

```
Property Tree Widget:
  Expanded:
    - /Global Options1
    - /Status1
    - /Axes1
    - /Marker1
    - /Marker2
    - /Map1
    - /Axes2
    - /LaserScan1
  Splitter Ratio: 0.5
  Tree Height: 607
- Class: rviz/Selection
  Name: Selection
- Class: rviz/Tool Properties
  Expanded:
    - /2D Pose Estimate1
    - /2D Nav Goal1
    - /Publish Point1
  Name: Tool Properties
  Splitter Ratio: 0.588679016
- Class: rviz/Views
  Expanded:
    - /Current View1
  Name: Views
  Splitter Ratio: 0.5
- Class: rviz/Time
  Experimental: false
  Name: Time
  SyncMode: 0
  SyncSource: ""
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz/Grid
      Color: 160; 160; 164
      Enabled: true
      Line Style:
        Line Width: 0.0299999993
        Value: Lines
      Name: Grid
      Normal Cell Count: 0
      Offset:
        X: 0
        Y: 0
        Z: 0
```

```
Plane: XY
Plane Cell Count: 10
Reference Frame: <Fixed Frame>
Value: true
- Class: rviz/Axes
  Enabled: true
  Length: 2
  Name: Axes
  Radius: 0.200000003
  Reference Frame: world
  Value: true
- Class: rviz/Marker
  Enabled: true
  Marker Topic: /visualization_marker_pose
  Name: Marker
  Namespaces:
    "": true
  Queue Size: 100
  Value: true
- Class: rviz/Marker
  Enabled: true
  Marker Topic: /visualization_marker_ref
  Name: Marker
  Namespaces:
    "": true
  Queue Size: 100
  Value: true
- Alpha: 0.699999988
  Class: rviz/Map
  Color Scheme: map
  Draw Behind: false
  Enabled: true
  Name: Map
  Topic: /map
  Unreliable: false
  Use Timestamp: false
  Value: true
- Class: rviz/Axes
  Enabled: true
  Length: 1.20000005
  Name: Axes
  Radius: 0.200000003
  Reference Frame: base_pose_ground_truth
  Value: true
- Alpha: 1
  Autocompute Intensity Bounds: true
  Autocompute Value Bounds:
    Max Value: 10
```

```
    Min Value: -10
    Value: true
  Axis: Z
  Channel Name: intensity
  Class: rviz/LaserScan
  Color: 255; 255; 255
  Color Transformer: Intensity
  Decay Time: 0
  Enabled: true
  Invert Rainbow: false
  Max Color: 255; 255; 255
  Max Intensity: -999999
  Min Color: 0; 0; 0
  Min Intensity: 999999
  Name: LaserScan
  Position Transformer: XYZ
  Queue Size: 10
  Selectable: true
  Size (Pixels): 3
  Size (m): 0.200000003
  Style: Flat Squares
  Topic: /base_scan
  Unreliable: false
  Use Fixed Frame: true
  Use rainbow: true
  Value: true
Enabled: true
Global Options:
  Background Color: 48; 48; 48
  Default Light: true
  Fixed Frame: world
  Frame Rate: 30
Name: root
Tools:
  - Class: rviz/Interact
    Hide Inactive Objects: true
  - Class: rviz/MoveCamera
  - Class: rviz/Select
  - Class: rviz/FocusCamera
  - Class: rviz/Measure
  - Class: rviz/SetInitialPose
    Topic: /initialpose
  - Class: rviz/SetGoal
    Topic: /move_base_simple/goal
  - Class: rviz/PublishPoint
    Single click: true
    Topic: /clicked_point
Value: true
```

Views:

Current:

```
Class: rviz/Orbit
Distance: 97.604599
Enable Stereo Rendering:
  Stereo Eye Separation: 0.0599999987
  Stereo Focal Distance: 1
  Swap Stereo Eyes: false
  Value: false
Focal Point:
  X: -1.71382999
  Y: 13.2875004
  Z: -25.2273006
Focal Shape Fixed Size: true
Focal Shape Size: 0.0500000007
Invert Z Axis: false
Name: Current View
Near Clip Distance: 0.00999999978
Pitch: 0.809800267
Target Frame: <Fixed Frame>
Value: Orbit (rviz)
Yaw: 4.46677637
```

Saved: ~

Window Geometry:

Displays:

```
  collapsed: false
Height: 848
Hide Left Dock: false
Hide Right Dock: true
QMainWindow State:
000000ff00000000fd000000040000000000000016a
000002eefc0200000008fb0000001200530065006c
0065006300740069006f006e00000001e10000009b
0000006400ffffffb0000001e0054006f006f006c
002000500072006f00700065007200740069006500
7302000001ed000001df00000185000000a3fb0000
00120056006900650077007300200054006f006f02
000001df000002110000018500000122fb00000020
0054006f006f006c002000500072006f0070006500
720074006900650073003203000002880000011d00
0002210000017afb00000010004400690073007000
6c0061007900730100000000000002ee000000dd00
fffffffb0000002000730065006c00650063007400
69006f006e00200062007500660066006500720200
000138000000aa0000023a00000294fb0000001400
5700690064006500530074006500720065006f0200
0000e6000000d2000003ee0000030bfb0000000c00
4b0069006e00650063007402000001860000010600
```

```

00030c00000261000000010000010f00000386fc02
00000003fb0000001e0054006f006f006c00200050
0072006f0070006500720074006900650073010000
0041000000780000000000000000fb0000000a0056
0069006500770073000000000000000386000000b0
00fffffffb0000001200530065006c006500630074
0069006f006e010000025a000000b2000000000000
0000000000200000490000000a9fc0100000001fb
0000000a00560069006500770073030000004e0000
0080000002e1000001970000000300000500000000
3efc0100000002fb0000000800540069006d006501
00000000000005000000030000fffffffb00000008
00540069006d006501000000000000045000000000
0000000000000390000002ee000000040000000400
00000800000008fc00000001000000020000000100
00000a0054006f006f006c00730000000000ffffff
ff0000000000000000
Selection:
  collapsed: false
Time:
  collapsed: false
Tool Properties:
  collapsed: false
Views:
  collapsed: true
Width: 1280
X: 65
Y: 24

```

Por fim, é necessário criar um outro nó - ou adaptar um já existente - para enviar marcadores ao RViz. O novo nó pode ser feito da seguinte forma:

rvizsend.py

```

1 #!/usr/bin/env python
2 import rospy
3 from geometry_msgs.msg import Twist
4 from nav_msgs.msg import Odometry
5 from math import sqrt, atan2, exp, atan, cos, sin, acos, pi
6   , asin, atan2
7 from tf.transformations import euler_from_quaternion,
8   quaternion_from_euler
9 from time import sleep
10 from visualization_msgs.msg import Marker, MarkerArray
11 import tf
12 import sys
13
14 # Frequencia de simulacao no gazebo
15 global freq

```



```
14 freq = 20.0 # Hz
15
16 # Estados do robo
17 global x_n, y_n, theta_n
18 x_n = 0.1 # posicao x atual do robo
19 y_n = 0.2 # posicao y atual do robo
20 theta_n = 0.001 # orientacao atual do robo
21
22 # Rotina callback para a obtencao da pose do robo
23 def callback_pose(data):
24     global x_n, y_n, theta_n
25
26     x_n = data.pose.pose.position.x # posicao 'x' do robo
27         no mundo
28     y_n = data.pose.pose.position.y # posicao 'y' do robo
29         no mundo
30     x_q = data.pose.pose.orientation.x
31     y_q = data.pose.pose.orientation.y
32     z_q = data.pose.pose.orientation.z
33     w_q = data.pose.pose.orientation.w
34     euler = euler_from_quaternion([x_q, y_q, z_q, w_q])
35     theta_n = euler[2] # orientacao 'theta' do robo no
36         mundo
37
38     # Atualiza uma transformacao rigida entre os sistemas de
39     # coordenadas do mundo e do robo
40     # Necessario para o rviz apenas
41     br = tf.TransformBroadcaster()
42     br.sendTransform((x_n, y_n, 0), (x_q, y_q, z_q, w_q),
43         rospy.Time.now(), "/base_pose_ground_truth", "world"
44     )
45
46     return
47
48 # Rotina primaria
49 def example():
50     global freq
51     global x_n, y_n, theta_n
52     global pub_rviz_ref, pub_rviz_pose
53     i = 0
54     rospy.init_node("rviznode") # inicializa o no "este no"
55     rospy.Subscriber("/odom", Odometry, callback_pose) #
56         declaracao do topico onde sera lido o estado do robo
57
58     # Inicializa os nos para enviar os marcadores para o
59     # rviz
60     pub_rviz_ref = rospy.Publisher("/
```

```

        visualization_marker_ref", Marker, queue_size=1) #
        rviz marcador de velocidade de referencia
54 pub_rviz_pose = rospy.Publisher("/
        visualization_marker_pose", Marker, queue_size=1) #
        rviz marcador de velocidade do robo
55
56 # Define uma variavel que controlar[a a frequencia de
        execucao deste no
57 rate = rospy.Rate(freq)
58
59 sleep(0.2)
60
61 # O programa do no consiste no codigo dentro deste
        while
62 while not rospy.is_shutdown(): #"Enquanto o programa
        nao ser assassinado "
63
64     # Incrementa o tempo
65     i = i + 1
66     time = i / float(freq)
67     # Necessario para o rviz apenas
68     br = tf.TransformBroadcaster()
69     br.sendTransform((0, 0, 0), (0, 0, 0, 1), rospy.
        Time.now(), "/map", "world")
70     br = tf.TransformBroadcaster()
71     br.sendTransform((0, 0, 0), (0, 0, 0, 1), rospy.
        Time.now(), "base_laser_link", "
        base_pose_ground_truth")
72     # Espera por um tempo de forma a manter a frequencia
        desejada
73     rate.sleep()
74
75 # Funcao inicial
76 if __name__ == '__main__':
77     try:
78         example()
79     except rospy.ROSInterruptException:
80         pass

```

Ao invés de um novo nó, o nó de controle do robô pode ser incrementado da seguinte forma:

example_node2.py

```

1 #!/usr/bin/env python
2 import rospy
3 from geometry_msgs.msg import Twist
4 from nav_msgs.msg import Odometry
5 from math import sqrt, atan2, exp, atan, cos, sin, acos, pi
    , asin, atan2

```

```
6 from tf.transformations import euler_from_quaternion,
   quaternion_from_euler
7 from time import sleep
8 from visualization_msgs.msg import Marker, MarkerArray
9 import tf
10 import sys
11
12 # Frequencia de simulacao no stage
13 global freq
14 freq = 20.0 # Hz
15 # Velocidade de saturacao
16 global Usat
17 Usat = 5
18 # Estados do robo
19 global x_n, y_n, theta_n
20 x_n = 0.1 # posicao x atual do robo
21 y_n = 0.2 # posicao y atual do robo
22 theta_n = 0.001 # orientacao atual do robo
23 # Estados do robo
24 global x_goal, y_goal
25 x_goal = 0
26 y_goal = 0
27 # Relativo ao feedback linearization
28 global d
29 d = 0.80
30 # Relativo ao controlador (feedforward + ganho proporcional
   )
31 global Kp
32 Kp = 1
33
34 # Rotina callback para a obtencao da pose do robo
35 def callback_pose(data):
36     global x_n, y_n, theta_n
37     x_n = data.pose.pose.position.x # posicao 'x' do robo
   no mundo
38     y_n = data.pose.pose.position.y # posicao 'y' do robo
   no mundo
39     x_q = data.pose.pose.orientation.x
40     y_q = data.pose.pose.orientation.y
41     z_q = data.pose.pose.orientation.z
42     w_q = data.pose.pose.orientation.w
43     euler = euler_from_quaternion([x_q, y_q, z_q, w_q])
44     theta_n = euler[2] # orientaco 'theta' do robo no
   mundo
45     # Atualiza uma transformacao rigida entre os sistemas de
   coordenadas do mundo e do robo
46     #Necessario para o rviz apenas
47     br = tf.TransformBroadcaster()
```

```

48     br.sendTransform((x_n, y_n, 0), (x_q, y_q, z_q, w_q),
49                     rospy.Time.now(), "/base_pose_ground_truth", "world"
50                     )
51     return
52
51 # Rotina para a geracao da trajetoria de referencia
52 def reference_trajectory(time):
53     ##MUDAR PARA CIRCULO
54     global x_goal, y_goal
55     x_ref = x_goal
56     y_ref = y_goal
57     Vx_ref = 0
58     Vy_ref = 0
59     return (x_ref, y_ref, Vx_ref, Vy_ref)
60
61 # Rotina para a geracao da entrada de controle
62 def trajectory_controller(x_ref, y_ref, Vx_ref, Vy_ref):
63     global x_n, y_n, theta_n
64     global Kp
65     global Usat
66     Ux = Vx_ref + Kp * (x_ref - x_n)
67     Uy = Vy_ref + Kp * (y_ref - y_n)
68     absU = sqrt(Ux ** 2 + Uy ** 2)
69     if (absU > Usat):
70         Ux = Usat * Ux / absU
71         Uy = Usat * Uy / absU
72     return (Ux, Uy)
73
74 # Rotina feedback linearization
75 def feedback_linearization(Ux, Uy):
76     global x_n, y_n, theta_n
77     global d
78     VX = cos(theta_n) * Ux + sin(theta_n) * Uy
79     WZ = (-sin(theta_n) / d) * Ux + (cos(theta_n) / d) * Uy
80     return (VX, WZ)
81
82 # Rotina para publicar informacoes no rviz
83 def send_marker_to_rviz(x_ref, y_ref, Ux, Uy):
84     global x_n, y_n, theta_n
85     global x_goal, y_goal
86     global pub_rviz_ref, pub_rviz_pose
87     #Inicializa mensagens do tipo Marker
88     mark_ref = Marker()
89     mark_vel = Marker()
90     # Define um marcador para representar o alvo
91     mark_ref.header.frame_id = "/world"
92     mark_ref.header.stamp = rospy.Time.now()
93     mark_ref.id = 0

```

```
94     mark_ref.type = mark_ref.SPHERE
95     mark_ref.action = mark_ref.ADD
96     mark_ref.scale.x = 0.5
97     mark_ref.scale.y = 0.5
98     mark_ref.scale.z = 0.5
99     mark_ref.color.a = 1.0
100    mark_ref.color.r = 1.0
101    mark_ref.color.g = 0.0
102    mark_ref.color.b = 0.0
103    mark_ref.pose.position.x = x_ref
104    mark_ref.pose.position.y = y_ref
105    mark_ref.pose.position.z = 0.25
106    mark_ref.pose.orientation.x = 0
107    mark_ref.pose.orientation.y = 0
108    mark_ref.pose.orientation.z = 0
109    mark_ref.pose.orientation.w = 1
110    # Define um marcador para representar a "velocidade
        desejada "
111    mark_vel.header.frame_id = "/world"
112    mark_vel.header.stamp = rospy.Time.now()
113    mark_vel.id = 1
114    mark_vel.type = mark_vel.ARROW
115    mark_vel.action = mark_vel.ADD
116    mark_vel.scale.x = 0.5 * sqrt(Ux ** 2 + Uy ** 2)
117    mark_vel.scale.y = 0.2
118    mark_vel.scale.z = 0.2
119    mark_vel.color.a = 0.5
120    mark_vel.color.r = 1.0
121    mark_vel.color.g = 1.0
122    mark_vel.color.b = 1.0
123    mark_vel.pose.position.x = x_n
124    mark_vel.pose.position.y = y_n
125    mark_vel.pose.position.z = 0.1
126    quaternion = quaternion_from_euler(0, 0, atan2(Uy,Ux))
127    mark_vel.pose.orientation.x = quaternion[0]
128    mark_vel.pose.orientation.y = quaternion[1]
129    mark_vel.pose.orientation.z = quaternion[2]
130    mark_vel.pose.orientation.w = quaternion[3]
131    #Publica os marcadores , que serao lidos pelo rviz
132    pub_rviz_ref.publish(mark_ref)
133    pub_rviz_pose.publish(mark_vel)
134    return
135
136 # Rotina primaria
137 def example():
138     global freq
139     global x_n, y_n, theta_n
140     global pub_rviz_ref, pub_rviz_pose
```

```

141     vel = Twist()
142     i = 0
143     pub_stage = rospy.Publisher("/cmd_vel", Twist,
        queue_size=1) #declaracao do topico para comando de
        velocidade
144     rospy.init_node("example_node") #inicializa o no "este
        no"
145     rospy.Subscriber("/base_pose_ground_truth", Odometry,
        callback_pose) #declaracao do topico onde sera lido
        o estado do robo
146     #Inicializa os nos para enviar os marcadores para o
        rviz
147     pub_rviz_ref = rospy.Publisher("/
        visualization_marker_ref", Marker, queue_size=1) #
        rviz marcador de velocidade de referencia
148     pub_rviz_pose = rospy.Publisher("/
        visualization_marker_pose", Marker, queue_size=1) #
        rviz marcador de velocidade do robo
149     #Define uma variavel que controlar[a a frequencia de
        execucao deste no
150     rate = rospy.Rate(freq)
151     sleep(0.2)
152
153     # O programa do no consiste no codigo dentro deste
        while
154     while not rospy.is_shutdown(): #"Enquanto o programa
        nao ser assassinado"
155         # Incrementa o tempo
156         i = i + 1
157         time = i / float(freq)
158         # Obtem a trajetoria de referencia "constante neste
        exemplo"
159         [x_ref, y_ref, Vx_ref, Vy_ref] =
            refference_trajectory(time)
160         # Aplica o controlador
161         [Ux, Uy] = trajectory_controller(x_ref, y_ref,
            Vx_ref, Vy_ref)
162         # Aplica o feedback linearization
163         [V_forward, w_z] = feedback_linearization(Ux, Uy)
164         # Publica as velocidades
165         vel.linear.x = V_forward
166         vel.angular.z = w_z
167         pub_stage.publish(vel)
168         # Atualiza uma transformacao rigida entre os
        sistemas de coordenadas do mapa e do mundo
169         # Necessario para o rviz apenas
170         br = tf.TransformBroadcaster()
171         br.sendTransform((0, 0, 0), (0, 0, 0, 1), rospy.

```

```

    Time.now(), "/map", "world")
172 br = tf.TransformBroadcaster()
173 br.sendTransform((0, 0, 0), (0, 0, 0, 1), rospy.
    Time.now(), "base_laser_link", "
    base_pose_ground_truth")
174 #Chama a funcao que envia os marcadores para o rviz
175 send_marker_to_rviz(x_ref, y_ref, Ux, Uy)
176 #Espera por um tempo de forma a manter a frequencia
    desejada
177 rate.sleep()
178
179 # Funcao inicial
180 if __name__ == '__main__':
181
182     # Obtem os argumentos , no caso a posicao do alvo
183     x_goal = int(sys.argv[1])
184     y_goal = int(sys.argv[2])
185
186
187     try:
188         example()
189     except rospy.ROSInterruptException:
190         pass

```

Por fim, ao executar os quatro nós, é possível observar a simulação também no RViz, Fig. 8.0.1.

```

$ rosrn stage_ros stageros -d ./src/example/wods/map_1.
world
$ rosrn rviz rviz -d ./src/example/rviz/rviz_cfig.rviz
$ rosrn map_server map_server ./src/example/ma/map_1.
yaml
$ rosrn example example_node2.py 0 0

```

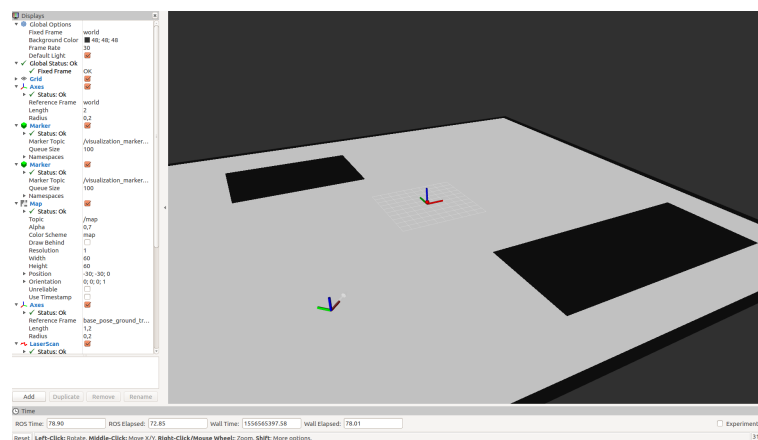
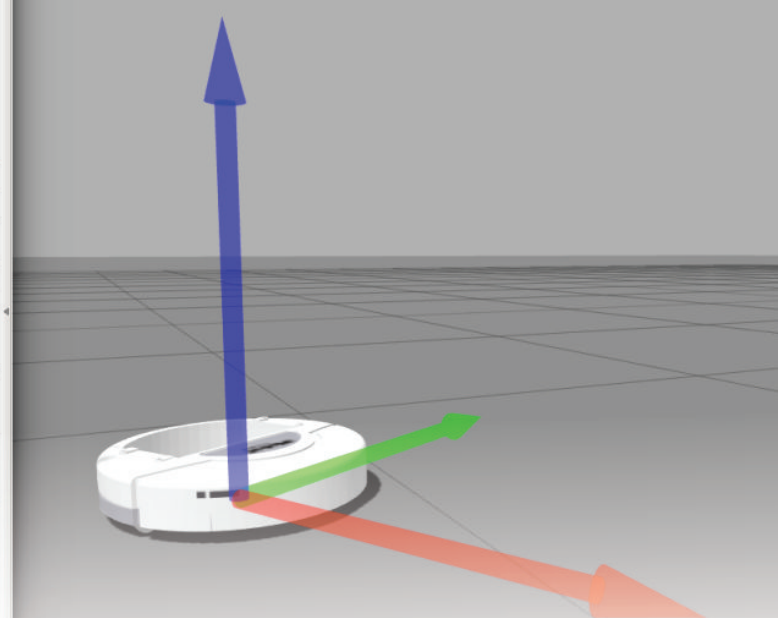
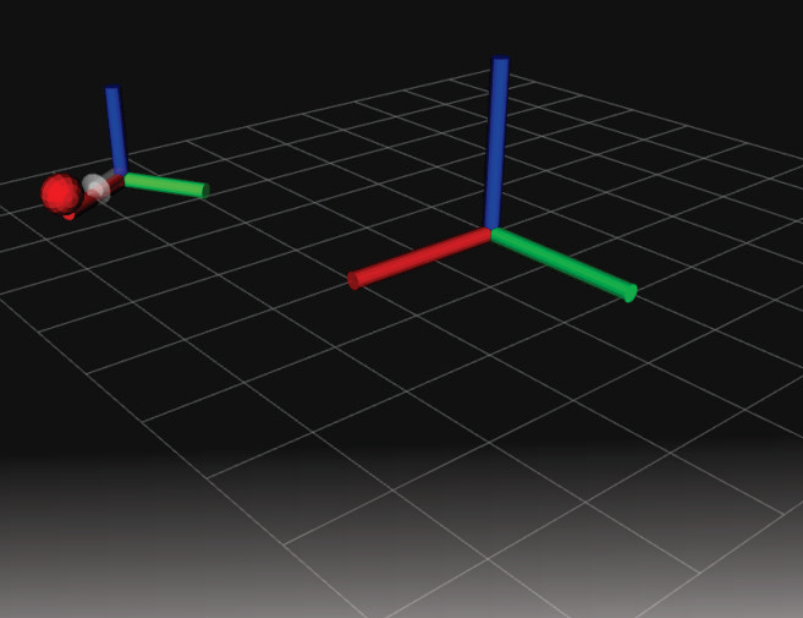


Figura 8.0.1: Visualização da Simulação Stage/RViz

Condensando em um arquivo *launch*:

example.launch

```
1 <?xml version="1.0"?>
2
3 <launch>
4
5 <!--Run the stage simulator-->
6 <node pkg = "stage_ros" name = "stageros" type = "stageros"
   output = "screen" args="-d $(find example)/worlds/map_1
   .world">
7 </node>
8
9 <!--Run the rviz for better visualization-->
10 <node pkg = "rviz" name = "rviz" type = "rviz" args="-d $(
   find example)/rviz/rviz_config.rviz" output="screen">
11 </node>
12
13 <!--Run the cntroller node
   args="x_goal
   y_goal" -->
14 <node pkg = "example" name = "example_node" type = "
   example_node2.py" args="0 0" output="screen">
15 </node>
16
17 <!--Run this only to allow the map to be shown in rviz-->
18 <node pkg = "map_server" name = "map_server" type = "
   map_server" args="$(find example)/maps/map_1.yaml"
   output="screen">
19 </node>
20
21 </launch>
```

9. Gazebo

O Gazebo é um importante simulador tridimensional que será coberto neste capítulo. Para abri-lo, pode-se abrir um "mundo vazio" que contém os itens plano infinito e iluminação, digitando em um terminal:

```
$ roslaunch gazebo_ros empty_world.launch
```

No canto superior esquerdo, alguns modelos podem ser encontrados e adicionados ao simulador, mas para efetivamente controlá-lo pelo ROS, é preciso definir um modelo do tipo *sdf* ou *urdf*, no qual se colocará os tópicos para subscrever e publicar.

Para o exemplo, será criado e configurado um novo pacote dentro da workspace usual.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg example_gazebo
$ cd example_gazebo
$ mkdir worlds
$ mkdir launch
$ mkdir scripts
```

O mundo vazio aberto anteriormente pode ser criado manualmente.

Para isso, deve-se criar dois arquivos, o primeiro, a configuração do mundo, dentro da pasta worlds:

first_world.world

```
1 <?xml version="1.0" ?>
2   <sdf version="1.5">
3     <world name="default">
```

```

4     <include >
5         <uri >model:// sun </uri >
6     </include >
7     <include >
8         <uri >model:// ground_plane </uri >
9     </include >
10    </world >
11 </sdf >

```

Outros modelos podem ser adicionados manualmente com a simulação aberta ou ainda no código entre as tags `<include>`.

Em seguida, deve-se criar um arquivo launch responsável por iniciar o mundo anteriormente configurado:

first_world1.launch

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <launch >
3     <arg name="debug" default="false" />
4     <arg name="gui" default="true" />
5     <arg name="pause" default="true" />
6     <arg name="world" default="$(find example_gazebo)/
7         worlds/first_world.world" />
8     <include file="$(find gazebo_ros)/launch/empty_world.
9         launch" >
10        <arg name="world_name" value="$(arg world)" />
11        <arg name="debug" value="$(arg debug)" />
12        <arg name="gui" value="$(arg gui)" />
13        <arg name="paused" value="$(arg pause)" />
14        <arg name="use_sim_time" value="true" />
15    </include >
16 </launch >

```

Feito isso, ao executar o arquivo launch, o mundo configurado com sol e com o plano infinito, isto é, igual ao mundo vazio `empty_world.launch` é aberto:

```
$ roslaunch example_gazebo first_world.launch
```

Para adicionar robôs na simulação, é necessário defini-lo em modelos `sdf` ou `urdf`. No exemplo será utilizado um modelo `sdf` do iRobot Create. Esse modelo encontra-se no repositório: https://github.com/joelillo/create_simulator/tree/master/model

É recomendado clonar a pasta para onde ficam os pacotes da workspace usual e ainda adicionar `CMakeList.txt` e `package.xml` a pasta para que o ROS a indentifique como pacote, e assim fique fácil encontrar seus arquivos por meio dos comandos do ROS.

Feito isso, é possível colocar o modelo no "mundo vazio" do gazebo pelos comandos:

(Note que será usado o modelo 1_4)

```
$ roslaunch gazebo_ros empty_world.launch
$ rosrun gazebo_ros spawn_model -file ./src/create/model
  -1_4.sdf -sdf -model my_create -x 0.0 -y 0.0
```

É crucial que o modelo tenha definido os tópicos que irá subscrever e publicar para que possa ser controlado pelo ROS. Portanto, é importante verificar se o arquivo *model-1_4.sdf* contém a seguinte definição no final:

```
1 <plugin name="differential_drive_controller" filename="
  libgazebo_ros_diff_drive.so">
2   <alwaysOn>true </alwaysOn>
3   <updateRate>${update_rate}</updateRate>
4   <leftJoint>right_wheel </leftJoint>
5   <rightJoint>left_wheel </rightJoint>
6   <wheelSeparation>0.5380</wheelSeparation>
7   <wheelDiameter>0.2410</wheelDiameter>
8   <torque>20</torque>
9   <commandTopic>cmd_vel </commandTopic>
10  <odometryTopic>odom</odometryTopic>
11  <odometryFrame>odom</odometryFrame>
12  <robotBaseFrame>base_footprint </robotBaseFrame>
13 </plugin>
```

Este código define que o robô irá subscrever no tópico *cmd_vel* e publicar no tópico *odom*. Agora, basta fazer o nó que controlará o robô. Para isso, será utilizado uma adaptação do código que controlava a simulação no *Stage* e *RViz* da seção anterior.

example_gazebonode.py

```
1 #!/usr/bin/env python
2 import rospy
3 from geometry_msgs.msg import Twist
4 from nav_msgs.msg import Odometry
5 from math import sqrt, atan2, exp, atan, cos, sin, acos, pi
  , asin, atan2
6 from tf.transformations import euler_from_quaternion,
  quaternion_from_euler
7 from time import sleep
8 from visualization_msgs.msg import Marker, MarkerArray
9 import tf
10 import sys
11 global centro_x, centro_y, raio_x, raio_y, w
12 centro_x = 0
13 centro_y = 0
14 raio_x = 1
```

```

15 raio_y = 1
16 w = 0.1
17 # Frequencia de simulacao no gazebo
18 global freq
19 freq = 20.0 # Hz
20 # Velocidade de saturacao
21 global Usat
22 Usat = 10
23 # Estados do robo
24 global x_n, y_n, theta_n
25 x_n = 0.1 # posicao x atual do robo
26 y_n = 0.2 # posicao y atual do robo
27 theta_n = 0.001 # orientacao atual do robo
28 # Estados do robo
29 global x_goal, y_goal
30 x_goal = 0
31 y_goal = 0
32 # Relativo ao feedback linearization
33 global d
34 d = 0.20
35 # Relativo ao controlador (feedforward + ganho proporcional
    )
36 global Kp
37 Kp = 0.5
38
39 # Rotina callback para a obtencao da pose do robo
40 def callback_pose(data):
41     global x_n, y_n, theta_n
42     x_n = data.pose.pose.position.x # posicao 'x' do robo
        no mundo
43     y_n = data.pose.pose.position.y # posicao 'y' do robo
        no mundo
44     x_q = data.pose.pose.orientation.x
45     y_q = data.pose.pose.orientation.y
46     z_q = data.pose.pose.orientation.z
47     w_q = data.pose.pose.orientation.w
48     euler = euler_from_quaternion([x_q, y_q, z_q, w_q])
49     theta_n = euler[2] # orientaco 'theta' do robo no
        mundo
50     # Atualiza uma transformacao rigida entre os sistemas de
        coordenadas do mundo e do robo
51     #Necessario para o rviz apenas
52     br = tf.TransformBroadcaster()
53     br.sendTransform((x_n, y_n, 0), (x_q, y_q, z_q, w_q),
        rospy.Time.now(), "/base_pose_ground_truth", "world"
        )
54     return
55

```

```
56 # Rotina para a geracao da trajetoria de referencia
57 def refference_trajectory(time):
58     global raio_x, raio_y, w
59     x_ref = raio_x * cos(w*time) + centro_x
60     y_ref = raio_y * sin(w*time) + centro_y
61     Vx_ref = - raio_x*sin(w*time)*w
62     Vy_ref = raio_x*cos(w*time)*w
63     return (x_ref, y_ref, Vx_ref, Vy_ref)
64
65 # Rotina para a geracao da entrada de controle
66 def trajectory_controller(x_ref, y_ref, Vx_ref, Vy_ref):
67     global x_n, y_n, theta_n
68     global Kp
69     global Usat
70     Ux = Vx_ref + Kp * (x_ref - x_n)
71     Uy = Vy_ref + Kp * (y_ref - y_n)
72     absU = sqrt(Ux ** 2 + Uy ** 2)
73     if (absU > Usat):
74         Ux = Usat * Ux / absU
75         Uy = Usat * Uy / absU
76
77     return (Ux, Uy)
78
79 # Rotina feedback linearization
80 def feedback_linearization(Ux, Uy):
81     global x_n, y_n, theta_n
82     global d
83     VX = cos(theta_n) * Ux + sin(theta_n) * Uy
84     WZ = (-sin(theta_n) / d) * Ux + (cos(theta_n) / d) * Uy
85     return (VX, WZ)
86
87 # Rotina para publicar informacoes no rviz
88 def send_marker_to_rviz(x_ref, y_ref, Ux, Uy):
89     global x_n, y_n, theta_n
90     global x_goal, y_goal
91     global pub_rviz_ref, pub_rviz_pose
92     #Inicializa mensagens do tipo Marker
93     mark_ref = Marker()
94     mark_vel = Marker()
95     # Define um marcador para representar o alvo
96     mark_ref.header.frame_id = "/world"
97     mark_ref.header.stamp = rospy.Time.now()
98     mark_ref.id = 0
99     mark_ref.type = mark_ref.SPHERE
100    mark_ref.action = mark_ref.ADD
101    mark_ref.scale.x = 0.5
102    mark_ref.scale.y = 0.5
103    mark_ref.scale.z = 0.5
```

```

104     mark_ref.color.a = 1.0
105     mark_ref.color.r = 1.0
106     mark_ref.color.g = 0.0
107     mark_ref.color.b = 0.0
108     mark_ref.pose.position.x = x_ref
109     mark_ref.pose.position.y = y_ref
110     mark_ref.pose.position.z = 0.25
111     mark_ref.pose.orientation.x = 0
112     mark_ref.pose.orientation.y = 0
113     mark_ref.pose.orientation.z = 0
114     mark_ref.pose.orientation.w = 1
115     # Define um marcador para representar a "velocidade
        desejada"
116     mark_vel.header.frame_id = "/world"
117     mark_vel.header.stamp = rospy.Time.now()
118     mark_vel.id = 1
119     mark_vel.type = mark_vel.ARROW
120     mark_vel.action = mark_vel.ADD
121     mark_vel.scale.x = 0.5 * sqrt(Ux ** 2 + Uy ** 2)
122     mark_vel.scale.y = 0.2
123     mark_vel.scale.z = 0.2
124     mark_vel.color.a = 0.5
125     mark_vel.color.r = 1.0
126     mark_vel.color.g = 1.0
127     mark_vel.color.b = 1.0
128     mark_vel.pose.position.x = x_n
129     mark_vel.pose.position.y = y_n
130     mark_vel.pose.position.z = 0.1
131     quaternio = quaternion_from_euler(0, 0, atan2(Uy,Ux))
132     mark_vel.pose.orientation.x = quaternio[0]
133     mark_vel.pose.orientation.y = quaternio[1]
134     mark_vel.pose.orientation.z = quaternio[2]
135     mark_vel.pose.orientation.w = quaternio[3]
136     # Publica os marcadores , que serao lidos pelo rviz
137     pub_rviz_ref.publish(mark_ref)
138     pub_rviz_pose.publish(mark_vel)
139     return
140
141 # Rotina primaria
142 def example():
143     global freq
144     global x_n, y_n, theta_n
145     global pub_rviz_ref, pub_rviz_pose
146     global x_goal, y_goal, raio_x, raio_y, centro_x,
        centro_y, w
147     vel = Twist()
148     i = 0
149     pub_stage = rospy.Publisher("/cmd_vel", Twist,

```

```

        queue_size=1) #declaracao do topico para comando de
        velocidade
150   rospy.init_node("gaze_node") #inicializa o no "este no"
151   rospy.Subscriber("/odom", Odometry, callback_pose) #
        declaracao do topico onde sera lido o estado do robo
152   #Inicializa os nos para enviar os marcadores para o
        rviz
153   pub_rviz_ref = rospy.Publisher("/
        visualization_marker_ref", Marker, queue_size=1) #
        rviz marcador de velocidade de referencia
154   pub_rviz_pose = rospy.Publisher("/
        visualization_marker_pose", Marker, queue_size=1) #
        rviz marcador de velocidade do robo
155   #Define uma variavel que controlar[a a frequencia de
        execucao deste no
156   rate = rospy.Rate(freq)
157   sleep(0.2)
158
159   # O programa do no consiste no codigo dentro deste
        while
160   while not rospy.is_shutdown(): #"Enquanto o programa
        nao ser assassinado "
161       # Incrementa o tempo
162       i = i + 1
163       time = i / float(freq)
164       # Obtem a trajetoria de referencia "constante neste
        exemplo "
165       [x_ref, y_ref, Vx_ref, Vy_ref] =
        refference_trajectory(time)
166       #Aplica o controlador
167       [Ux, Uy] = trajectory_controller(x_ref, y_ref,
        Vx_ref, Vy_ref)
168       # Aplica o feedback linearization
169       [V_forward, w_z] = feedback_linearization(Ux, Uy)
170       # Publica as velocidades
171       vel.linear.x = V_forward
172       vel.angular.z = w_z
173       pub_stage.publish(vel)
174       # Atualiza uma transformacao rigida entre os
        sistemas de coordenadas do mapa e do mundo
175       # Necessario para o rviz apenas
176       br = tf.TransformBroadcaster()
177       br.sendTransform((0, 0, 0), (0, 0, 0, 1), rospy.
        Time.now(), "/map", "world")
178       br = tf.TransformBroadcaster()
179       br.sendTransform((0, 0, 0), (0, 0, 0, 1), rospy.
        Time.now(), "base_laser_link", "
        base_pose_ground_truth")

```

```

180     #Chama a funcao que envia os marcadores para o rviz
181     send_marker_to_rviz(x_ref, y_ref, Ux, Uy)
182     #Espera por um tempo de forma a manter a frequencia
        desejada
183     rate.sleep()
184
185 # Funcao inicial
186 if __name__ == '__main__':
187     # Obtem os argumentos , no caso a posicao do alvo
188     centro_x = int(sys.argv[1])
189     centro_y = int(sys.argv[2])
190     raio_x = int(sys.argv[3])
191     raio_y = int(sys.argv[4])
192     try:
193         example()
194     except rospy.ROSInterruptException:
195         pass

```

Nota-se que os procedimentos responsáveis por enviar os marcadores ao *RViz* continuam sendo utilizados. Desta forma, o Gazebo também pode ser utilizado em conjunto com o visualizador, desde que este seja iniciado com as configurações corretas, disponíveis no código *rviz_config.rviz*, também feito anteriormente, no pacote *example*.

Sendo assim, toda a simulação pode ser condensada em um arquivo *launch*.

example_gazebo.launch

```

1 <launch >
2
3 <include file="$(find gazebo_ros)/launch/empty_world.
    launch">
4 </include >
5
6 <node pkg = "rviz" name = "rviz" type = "rviz" args="-d $
    (find example)/rviz/rviz_config.rviz" output="screen">
7 </node>
8
9 <node pkg="gazebo_ros" type="spawn_model" name = "myrob"
    args="-file $(find create)/model-1_4.sdf -sdf -model
    create " >
10 </node>
11
12 <node pkg="gazeteste" type="gazenode.py" name="gazenode"
    args="0 0 5 5"/>
13 </launch >

```

Executando este último código, com todos os arquivos dentro das devidas pastas e pacotes, a simulação irá se parecer com a Fig. 9.0.1.


```
$ roslaunch example_gazebo example_gazebo.launch
```

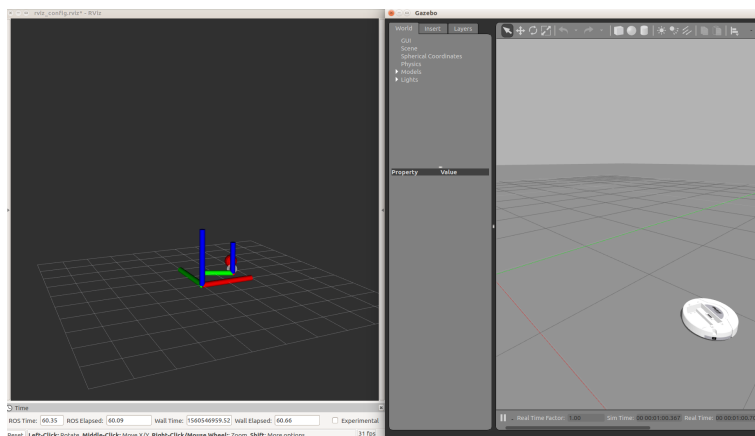
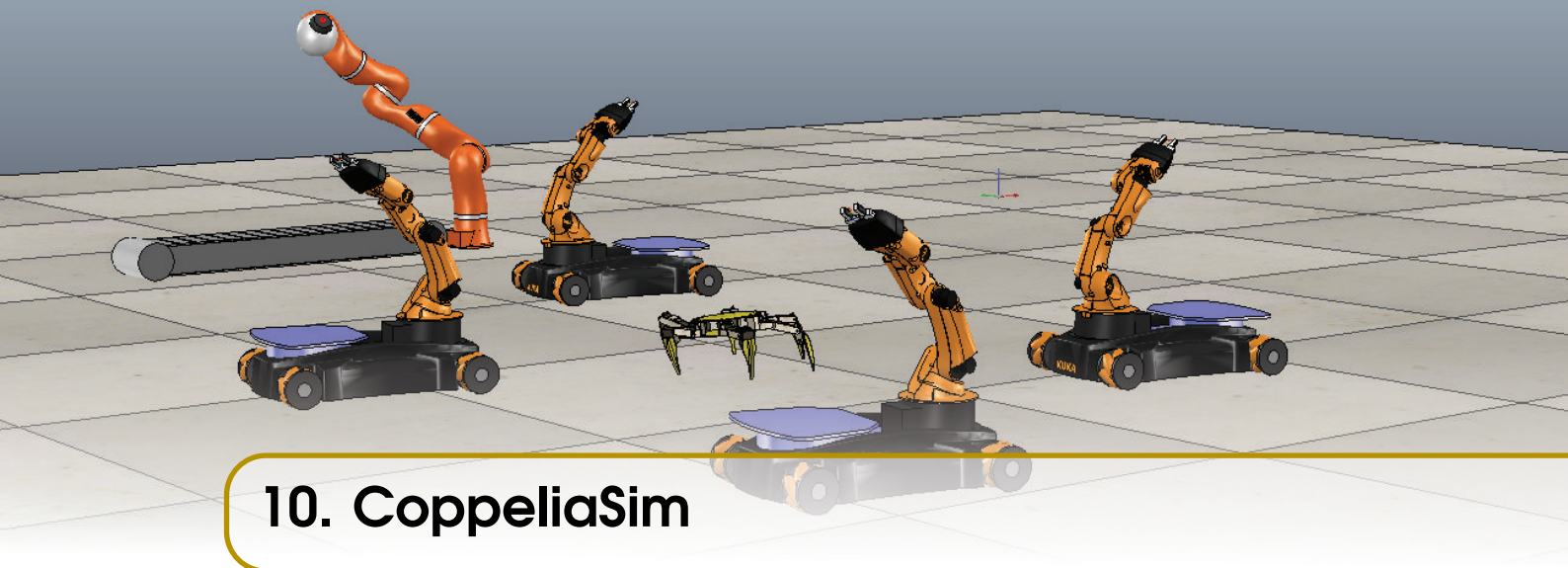


Figura 9.0.1: Execução da Simulação Gazebo e RViz



10. CoppeliaSim

10.1 Introdução

O CoppeliaSim, antes chamado de *Virtual Robot Experimentation Platform* ou **V-REP**, é um simulador, desenvolvido pela Coppelia Robotics. Possui ambiente de desenvolvimento integrado (IDE) e interface gráfica do usuário (GUI) para criar, compor e simular robôs. Há versão educacional gratuita, **PRO EDU**, disponível para Linux, Windows e MacOS em <http://www.coppeliarobotics.com/>. Apesar de, a priori, o simulador não ter relação com o ROS, ele é muito potente e pode realizar importantes experimentos. Ele pode ser usado sozinho ou em conjunto com outro programa via comunicação remota. Esta comunicação pode ser feita para que o simulador receba instruções de programas em C++, Python, Java dentre muitas outras linguagens de programação. As comunicações que serão destacadas aqui serão: Comunicação com o ROS via plugin e comunicação com o Matlab via API Remota.

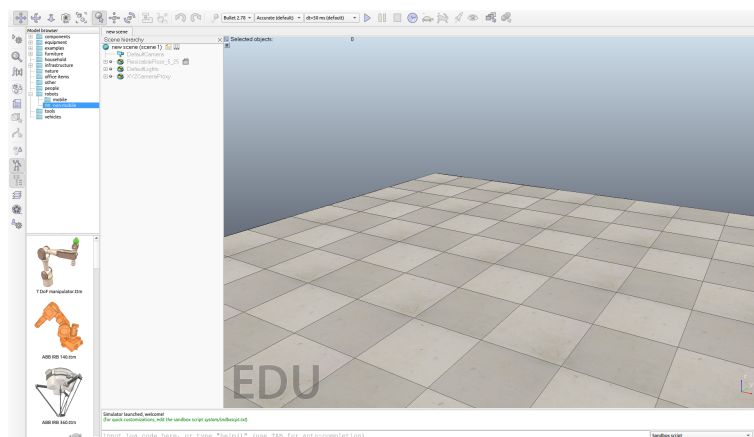


Figura 10.1.1: Execução do CoppeliaSim

Como o simulador pode ser instalado em qualquer sistema operacional, é necessário o

Linux apenas se você planeja utilizar o software em conjunto com o ROS.

10.2 Instalação

É recomendada a versão educacional gratuita: **PRO EDU**. Disponível em <http://www.coppeliarobotics.com/downloads.html>

A instalação é simples, para Windows basta prosseguir com o instalador. Para Linux, como geralmente há a necessidade de utilizar o simulador com o ROS, a instalação e configuração serão abordadas na seção de comunicação com o ROS, na página 109. Em resumo, basta descompactar a pasta em um repositório desejado e criar um *alias*.

10.3 Interface Gráfica do Usuário

Ao executar o programa pela primeira vez, observa-se um cenário como a da Fig. 10.1.1, detalhado na Fig. 10.3.1. Nele, há um chão e fundo já configurados.

Os comandos e atalhos da tela inicial são:

- 1 **Câmera.** Comandos que movimentam a câmera para facilitar a visualização do usuário. Basta selecionar o tipo de movimentação desejada e aplicá-la sobre a tela. Da esquerda para direita: o primeiro refere-se à translação da câmera, o segundo, à rotação, o terceiro ao afastamento ou aproximação, e os dois últimos à abertura e fechamento de ângulo para capturar toda a cena.
- 2 **Movimentação de Objetos.** Comandos para mover objetos em cena. Da esquerda para a direita: o primeiro botão é para a translação, o seguinte para a rotação, analogamente aos comandos de movimentação de câmera, enquanto que o terceiro associa objetos entre si e o último transfere características e propriedades entre objetos.
- 3 **Simulação.** Comandos para configurar a simulação. A primeira caixa de seleção, da esquerda para direita, mostra o módulo de simulação física e dinâmica escolhido para a simulação, contando com as opções *Bullet*, *ODE*, *Vortex* e *Newton*. A próxima, mostra a configuração escolhida para o módulo de simulação dinâmica do sistema, com opções *Very accurate*, *Accurate*, *Fast*, *Very fast* e *Customized*. A próxima é o passo da simulação - ou tempo de integração - quanto menor este tempo, mais preciso será o resultado mostrado, custando mais da máquina e deixando a simulação mais lenta. Mais à direita, há os botões que servem para iniciar, pausar e finalizar a simulação. O próximo botão seleciona a opção de simulação em tempo real. Os próximos dois comandam a dinâmica da simulação, tornando-a mais lenta ou mais rápida. O último botão seleciona a opção de melhoramento de threads.
- 4 **Barra de Ferramentas.** Principais comandos e atalhos para manipular objetos, scripts, propriedades e funções.
- 5 **Biblioteca** É onde encontram-se diversos modelos prontos de robôs e objetos, os quais podem ser utilizados para montar uma cena. Está subdividida em categorias, o que torna a utilização mais intuitiva.
- 6 **Hierarquia da Cena.** É onde são listados todos os componentes e objetos da cena, conforme sua estrutura hierárquica. O ícone de + indica que o componente possui subcomponentes associados. O ícone de papel escrito indica que o componente possui um *script* associado.

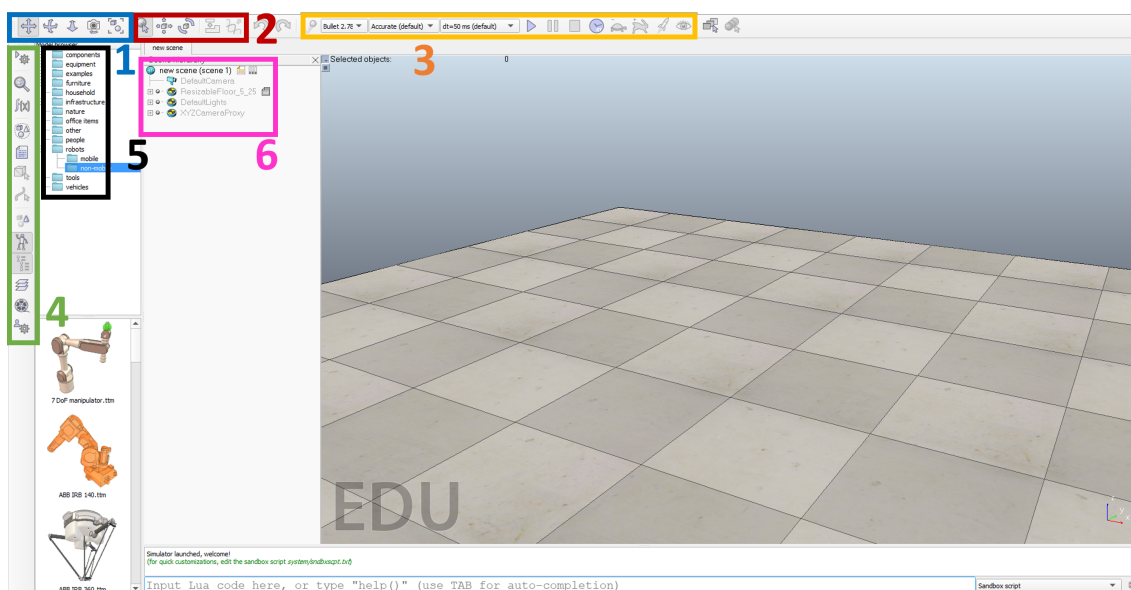


Figura 10.3.1: GUI do CoppeliaSim

10.4 Comunicação Remota

Uma importante vantagem deste simulador, é o fato de poder ser utilizado como cliente remoto utilizando rotinas com comunicação API (*Application Programming Interface*), geralmente em conjunto com ROS ou MATLAB. No entanto, as rotinas podem ser implementadas também em C++, Python, Java, Octave, Urbi e Lua - esta última, inclusive, é utilizada dentro do próprio programa. Para isso, é preciso estabelecer comunicações para enviar e receber dados.

10.4.1 ROS

Para comunicar com o ROS é necessário apenas que o programa possua um plugin em seu diretório de instalação. O plugin é geralmente chamado de *libv_repExtRosInterface.so*. Note que a extensão do Plugin é *.so*, logo, funciona apenas para sistemas operacionais Linux. Para poder usar o plugin em outros sistemas operacionais, deve-se realizar uma compilação cruzada. (O Windows, por exemplo, utiliza plugins com extensão *.dll*)

Para a compilação do plugin, será utilizado uma área de trabalho do tipo *catkin build*. Por conta disso, antes é necessário instalar alguns programas, caso ainda não tenha sido feito:

```
$ sudo apt-get install python-catkin-tools xsltproc ros-
$ROS_DISTRO-brics-actuator ros-$ROS_DISTRO-tf2-sensor-
msgs
```

Pode-se converter uma workspace do tipo *catkin_make* para *catkin build* utilizando os comandos:

```
$ catkin clean
$ catkin build
```

OBS: O comando *catkin clean* simplesmente apaga o conteúdo das pastas build e devel da workspace. Logo, esse procedimento pode ser feito manualmente.

Ele deve ser baixado em <http://coppeliarobotics.com/ubuntuVersions.html> Nessa apostila, foi usada a versão V-REP PRO EDU, Ubuntu 16.04.

Faça o download e descompacte a pasta em um diretório conveniente. Escolher um nome pequeno pode ser útil, pois a aplicação será comumente aberta via terminal. Sugere-se, portanto, nomear a pasta como *coppelia*.

OBS: Uma opção para descompactar arquivos *.tar* é usando comando similar a:

```
$ tar xf CoppeliaSim_Edu_V4_1_0_Ubuntu16_04.tar.xz
```

Deve-se adicionar ao *.bashrc* o caminho escolhido para o simulador: *export COPPELIASIM_ROOT_DIR= «caminho_para_a_root_do_coppelia»*.

```
$ echo "export COPPELIASIM_ROOT_DIR=\"<
caminho_para_a_root_do_coppelia>\"" >> ~/.bashrc
```

Atente-se para manter as aspas. Exemplo do comando com um possível caminho:

```
$ echo "export COPPELIASIM_ROOT_DIR=\"/home/arthur/
programs/coppelia\"" >> ~/.bashrc
```

Também é recomendado criar um *alias* para executar o simulador, ou seja, criar um comando que é um atalho para abrir o programa. Isso pode ser feito adicionando ao *.bashrc* o comando: *alias vrep=\$VREP_ROOT/vrep.sh*

```
echo "alias coppeliasim=$COPPELIASIM_ROOT_DIR/coppeliaSim
.sh" >> ~/.bashrc
```

Deve-se carregar as modificações feitas no *.bashrc* através do comando:

```
$ source ~/.bashrc
```

Ou, simplesmente, fechando e abrindo um novo terminal.

Verifique o funcionamento do programa que deve-se parecer com a Fig. ??, executando o comando:

```
$ coppeliasim
```

Deve-se agora navegar até à pasta *src* da workspace do tipo catkin build e clonar recursivamente um pacote chamado **simExtROSInterface**:

```
$ roscd catkin_ws/src
$ git clone --recursive https://github.com/
CoppeliaRobotics/simExtROSInterface.git
sim_ros_interface
```

OBS: Outros repositórios contendo o *vrep_ros_interface* podem ser usados, como por exemplo https://github.com/fayyazpocker/vrep_ros_interface.

Mais informações sobre como utilizar a interface pode ser encontrada no próprio link para o repositório no Github.

Recompile a área de trabalho:

```
$ catkin build
```

Caso tudo tenha ocorrido tudo bem até aqui, a biblioteca *libsimsExtROSInterface.so* já está compilada. Esta serve para fazer com que o simulador reconheça o ambiente ROS da máquina atual. Neste passo, deve-se copiar este plugin para o diretório de instalação do simulador, com o comando:

```
$ cp ~/catkin_ws/devel/lib/libsimsExtROSInterface.so
  $COPPELIASIM_ROOT_DIR
```

Agora deve-se testar a comunicação. Para isso, basta rodar primeiro o *master* do ROS e em seguida o Coppeliasim (ou V-REP):

```
$ roscore
# Em um novo terminal
$ coppeliasim
```

Nas mensagens de inicialização do simulador, a aparição das seguintes mensagens é um bom sinal de que a biblioteca foi compilada corretamente:

```
Plugin 'RosInterface': loading...
Plugin 'RosInterface': warning: replaced variable 'simROS
,
Plugin 'RosInterface': load succeeded.
```

Para confirmar a interação entre ROS e Coppeliasim (V-REP), após dar *play* na cena vazia que é aberta no início do programa, deve ser possível verificar a aparição do tópico */tf* no ROS:

```
$ rostopic list
/rosout
/rosout_agg
/tf
```

Caso apareça o tópico */tf*, bom sinal: a interface está habilitada e funcional.

Por padrão, a biblioteca *libsimsExtROSInterface.so* aceita a manipulação dos tipos comuns de mensagens do ROS. Para utilizar outros tipos de mensagens, ou até mesmo mensagens customizadas, deve-se fazer como indicado no repositório da interface.

10.4.2 MATLAB

Configurando

Em primeiro lugar, é preciso estabelecer o servidor. Este pode ser criado de duas formas:

- API Remota Contínua

Neste modo, os serviços estarão disponíveis mesmo quando a simulação não estiver sendo executada. Além disso, comandos como *play*, *pause* e *stop* podem ser executados.

Para isso, deve-se localizar o arquivo *remoteApiConnections.txt* dentro do diretório de instalação do simulador, geralmente C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu e alterar as três últimas linhas para:

```
portIndex1_port           = <PORTA>
portIndex1_debug          = false
portIndex1_syncSimTrigger = true
```

As portas 19997 e 19999 são as mais comuns de serem usadas neste caso.

- API Remota Temporária

Já neste modo, o próprio usuário deve controlar quando o servidor será inicializado ao der *play* na simulação. Os serviços só ficarão disponíveis enquanto a simulação estiver sendo executada, assim que ela for parada os serviços ficarão indisponíveis.

Para isso, em algum script dentro do simulador deve ser adicionado o comando:

```
simExtRemoteApiStart (<PORTA>)
```

Novamente, As portas 19997 e 19999 são as mais comuns.

Em segundo lugar, deve-se iniciar o cliente no Matlab. É preciso colocar no diretório corrente do Matlab alguns arquivos:

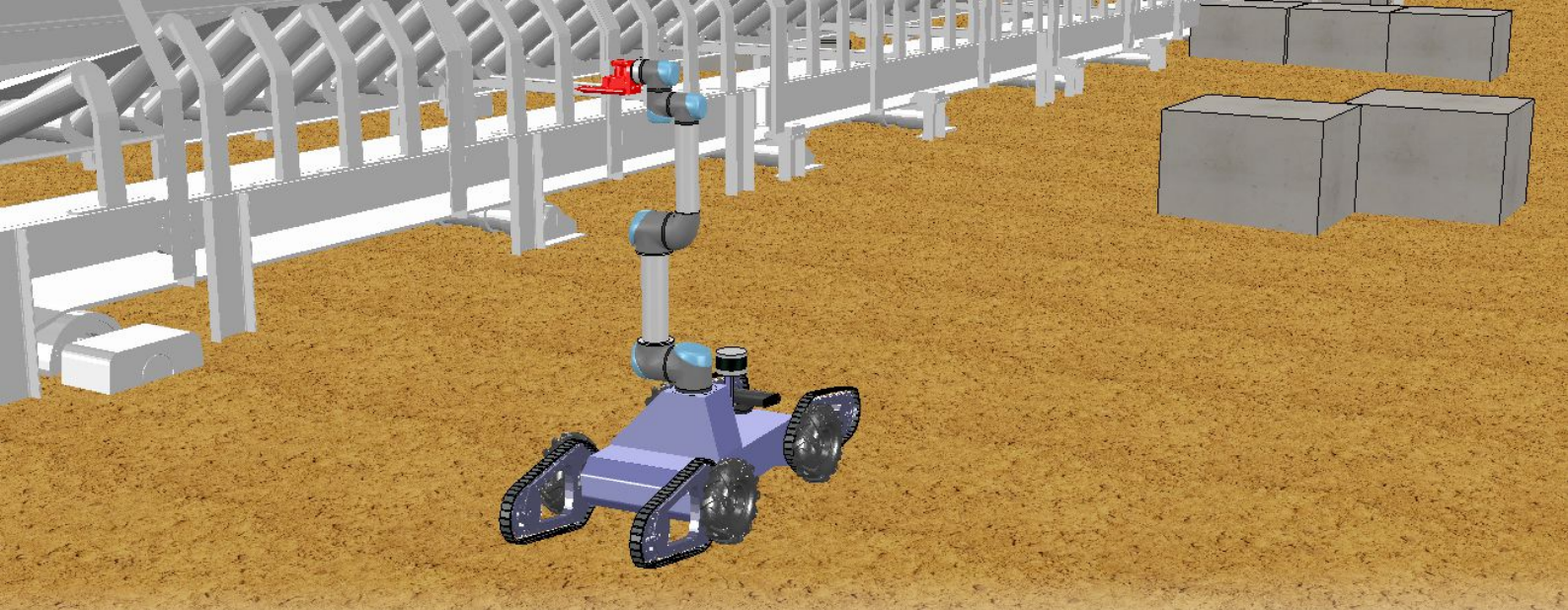
Localizados no diretório C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu\programming\remoteApiBi

Neste diretório é possível encontrar arquivos para interfaces com outras linguagens também. Por hora, são interessantes os arquivos *remApi.m*, *remoteApiProto.m* e *simpleTest.m*, dentro de matlab e o arquivo de biblioteca dentro de lib, que irá variar dependendo do sistema operacional, para Windows será *remoteApi.dll*.

Com os quatro dentro do diretório corrente do Matlab, o seguinte comando deve ser utilizado:

```
1 vrep=remApi('remoteApi');
```

Este comando cria um objeto, no qual se instanciarão todas as funções da API.



11. Teleop ROSI

Este exemplo usará o simulador da competição ROSI Challenge, www.sbai2019.com.br/rosi-challenge, disponível em github.com/filRocha/rosiChallenge-sbai2019. O simulador é um cenário que contém um Robô para Serviços de Inspeção (ROSI). A cena foi montada no CoppeliaSim e possui interface com o ROS. O objetivo deste exemplo é usar o nó `turtle_teleop_key` do pacote `turtlesim` para comandar o ROSI.

11.1 Baixo Nível

Para isso é preciso fazer um nó que receba a mensagem *Twist* do nó *teleop* e convertê-lo em velocidades para as rodas.

vel_to_wheels.py

```
1 #!/usr/bin/env python
2 #
3 # CODIGO QUE CONVERTE SINAIS DE VELOCIDADE EM COMANDOS PARA
4 # AS RODAS #
5 # Este nó representa o controle de baixo nível
6
7 import rospy
8 # Tipos de mensagens utilizadas
9 from rosi_defy.msg import RosiMovement, RosiMovementArray
10 from geometry_msgs.msg import Twist
11
```

```

12 # Classe RosiNodeClass responsavel por todo o processo do
    programa
13 class RosiNodeClass():
14
15     # Atributos segundo o manual
16     max_translational_speed = 5 # in [m/s]
17     max_rotational_speed = 10 # in [rad/s]
18     var_lambda = 0.965
19     wheel_radius = 0.1324
20     ycir = 0.531
21
22     # Distancia entre as rodas direitas e esquerdas
23     L = 0.5
24
25     # Construtor
26     def __init__(self):
27
28         # Comandos que serao enviados para as rodas
            da direita e da esquerda
29         self.omega_left = 0
30         self.omega_right = 0
31
32         # Atalhos para informar que deve mandar
            velocidade 0 quando nao houver comandos
            de velocidade
33         self.last = 0
34         self.TIME_OUT = 0.1
35
36         # Mensagem de inicializacao
37         rospy.loginfo('Controle de baixo nivel
            iniciado')
38
39         # Publicar em command_traction_speed
40         self.pub_traction = rospy.Publisher('/rosi/
            command_traction_speed',
            RosiMovementArray, queue_size=1)
41         # Subscrever em aai_rosi_cmd_vel
42         self.sub_cmd_vel = rospy.Subscriber('/
            aai_rosi_cmd_vel', Twist, self.
            callback_cmd_vel)
43         # Frequencia de publicacao
44         node_sleep_rate = rospy.Rate(10)
45
46         # Loop principal , responsavel pelos
            procedimentos chaves do programa
47         while not rospy.is_shutdown():
48
49             # Comando de tracao a ser publicado

```

```
50         traction_command_list =  
51             RosiMovementArray()  
52  
53         # Criar traction_command_list como  
54         # uma soma de traction_commands  
55         for i in range(4):  
56             # Um comando de tracao por  
57             # roda  
58             traction_command =  
59                 RosiMovement()  
60             # ID da roda  
61             traction_command.nodeID = i  
62             # Publicar 0 caso nao haja  
63             # comandos de velocidade  
64             if rospy.get_rostime().  
65                 to_sec() - self.last >=  
66                 self.TIME_OUT:  
67                 traction_command.  
68                 joint_var = 0  
69             else:  
70                 # Separa as rodas  
71                 # do lado direito  
72                 # do esquerdo  
73                 if i < 2:  
74                     traction_command  
75                         .  
76                         joint_var  
77                         = self.  
78                         omega_right  
79                 else:  
80                     traction_command  
81                         .  
82                         joint_var  
83                         = self.  
84                         omega_left  
85  
86             # Adiciona o comando ao  
87             # comando de tracao final  
88             traction_command_list.  
89             movement_array.append(  
90                 traction_command)  
91  
92         # Publicacao  
93         self.pub_traction.publish(  
94             traction_command_list)
```

```

73         # Pausa
74         node_sleep_rate.sleep()
75
76     # Callback da leitura do cmd_vel
77     def callback_cmd_vel(self, msg):
78
79         self.omega_right = (msg.linear.x + (self.L
80                               /2)*msg.angular.z)/self.wheel_radius
81         self.omega_left = (msg.linear.x - (self.L
82                               /2)*msg.angular.z)/self.wheel_radius
83
84         self.last = rospy.get_rostime().to_sec()
85
86 # Funcao principal
87 if __name__ == '__main__':
88
89     # Inicializa o no
90     rospy.init_node('rosi_vel_to_wheels_node',
91                     anonymous=True)
92
93     # Inicializa o objeto
94     try:
95         node_obj = RosiNodeClass()
96     except rospy.ROSInterruptException:
97         pass

```

11.2 Launch

Depois disso, basta fazer um arquivo de *launch* e nele realizar o *remap* necessário para o nó *teleop* enquanto que também executa o nó *vel_to_wheels*.

rosi_teleop_v2.launch

```

1 <launch>
2     <!-- No que transforma comandos do teclado em
3         comandos de velocidade -->
4     <node name="vel_to_wheels" pkg="aai_robotics" type="
5         vel_to_wheels.py" output = "screen" respawn="
6         true" />
7
8     <!-- No que le o teclado -->
9     <node pkg="turtlesim" name="teleop_key" type="
10        turtle_teleop_key">
11         <remap from="turtle1/cmd_vel" to="
12             aai_rosi_cmd_vel"/>
13     </node>
14
15     <!-- Parametros da simulacao para rodar mais leve
16     -->

```

```
11     <rosparam command="load" file="$(find rosi_defy)/  
12     config/simulation_parameters.yaml" />  
</launch>
```

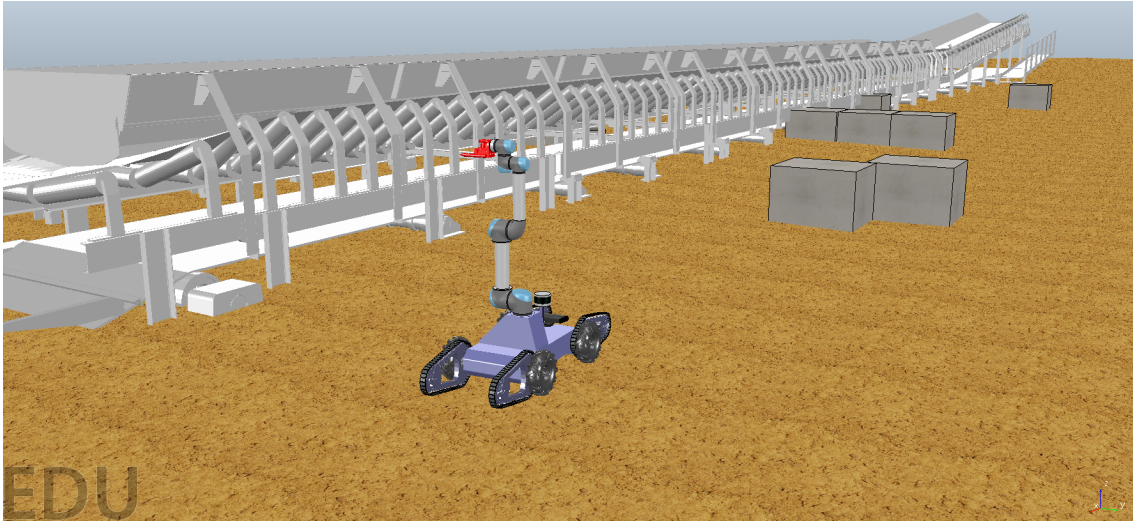


Figura 11.2.1: *Simulação do Teleop ROSI*



12. Cenário de um Manipulador

Este exemplo cobrirá a construção de um cenário para o controle de um manipulador no CoppeliaSim pelo Matlab, usando uma API Remota.

Antes de efetivamente controlar a simulação, é preciso criar o cenário. Trata-se de um arquivo *.ttx* com as configurações de objetos e itens utilizados.

Os passos a seguir, para a construção do cenário, devem ser feitos usando um cenário modelo já pronto. O cenário a ser utilizado encontra-se dentro do diretório padrão de instalação do CoppeliaSim e dentro de *scenes* deve-se localizar o arquivo **remoteApiCommandServerExample.ttx**. Recomenda-se criar uma cópia dele no diretório no qual será trabalhado no Matlab.

Este modelo de cenário é crucial para o comando, pois ele possui o objeto *remoteApiCommandServer*, necessário para o controle via Matlab.

OBS :

Para o controle de um manipulador, tem-se que: Variáveis:

- q : ângulo das juntas
- x : vetor de pose do efetuador (último link da cadeia cinemática)

Relações:

- Cinemática direta: $x = f_{CD}(q)$, onde f_{CD} é a função de cinemática direta, usualmente obtida a partir dos parâmetros de Denavit-Hatemberg.

- Cinemática inversa: $q = f_{CI}(x)$, onde f_{CI} é a inversa de f_{CD} , num caso geral é uma função muito dificilmente encontrada.

- Cinemática direta diferencial: $\dot{x} = J(q)\dot{q}$, onde $J(q)\dot{q} = \frac{\partial f_{CD}}{\partial q}$, ou seja, a Jacobiana de f_{CD} .

- Cinemática inversa diferencial: $\dot{q} = J^+(q)\dot{x}$, onde $J^+(q)$ é a pseudo inversa de $J(q)$, definida por:

$$J^+(q) = \begin{cases} [J(q)^T J(q)]^{-1} J(q)^T & \text{se } N_x > N_q, \\ J(q)^{-1} & \text{se } N_x = N_q, \\ J(q)^T [J(q) J(q)^T]^{-1} & \text{se } N_x < N_q, \end{cases} \quad (12.0.1)$$

onde N_x é o numero de graus de liberdade ta tarefa do manipulador e N_q é o número de juntas. Por exemplo, se um manipulador de 6 juntas deve seguir uma curva em 3D com uma orientação qualquer, temos $N_x = 3$ e $N_q = 6$ (caso sobre atuado). Caso a orientação seja especificada temos $N_x = 6$ e $N_q = 6$. Caso um manipulador de 2 juntas necessite seguir uma curva em 3D, temos $N_x = 3$ e $N_q = 2$ (caso sub atuado).

12.1 Cenário

A cena a seguir foi criada sem a utilização de modelos, para servir como exemplo. Mas para o controle do robô, deve-se usar uma cópia da cena-modelo **remoteApiCommandServerExample.ttt**, como mencionado acima. Neste modo, o controle será por API Remota Temporária. No objeto *remoteApiCommandServer* há a porta que será utilizada. Por padrão, será a 19999, caso o usuário deseje outra, deverá alterá-la.

Antes de adicionar objetos, a configuração do módulo de simulação dinâmica, presente no bloco 3 Fig. 10.3.1, será alterada para *Very accurate*. Dentro da biblioteca, bloco 5 Fig. 10.3.1, será selecionado, dentro de *furniture > tables*, o objeto *customizable table*, e este será solto na cena. Na tela aberta para ajustar o tamanho da mesa, os parâmetros serão alterados: *Lenght* para 1, *Widht* para 0.75 e *Height* para 0.5. A mesa pode ser rodada selecionando a opção *Object/Item Rotation/Orientation*, no bloco 2 Fig. 10.3.1, em seguida, selecionando *Rotation*, deve selecionar em *Relative to* para *Own frame* e modificar *Around Z[deg]* para 90 e clicar em *Z-rotate sel..* Ficará como a Fig. 12.1.1.

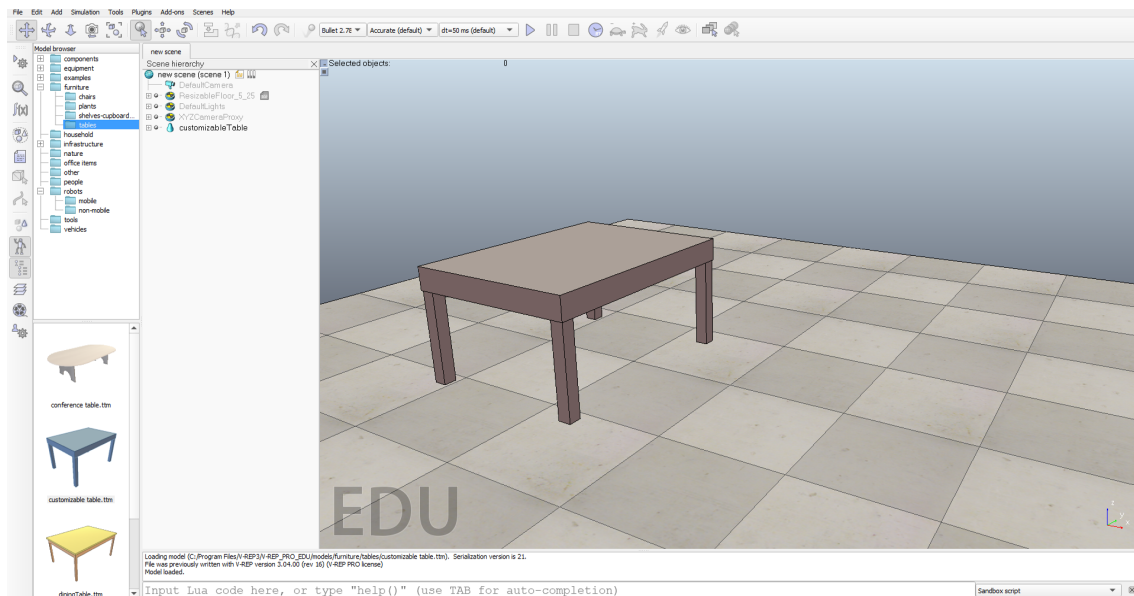


Figura 12.1.1: Passo 1

Novamente na biblioteca, será selecionado *resizable concret block* dentro de *infrastructure* > *other*. É necessário, agora, posicionar a mesma a um metro de distância da mesa. Para isso, deve-se selecionar ambos, usando a tecla shift, e clicando em *Object/Item Shift*, no bloco 2 Fig. 10.3.1. Pressionando, dentro de *Position*, os botões *Apply X to sel.* e *Apply Y to sel.*, o bloco se deslocará para dentro da mesa. Em seguida, selecionando apenas o bloco de concreto e novamente em *Object/Item Shift*, agora em *Translation*, marcando *Own frame* em *Relative to*, deve-se colocar o valor de 1 para *Along X[m]* e clicando em *X-translate sel.*. Ao final deste passo, o bloco deve estar a uma distância de um metro da mesa. Ficará similar a Fig. 12.1.2.

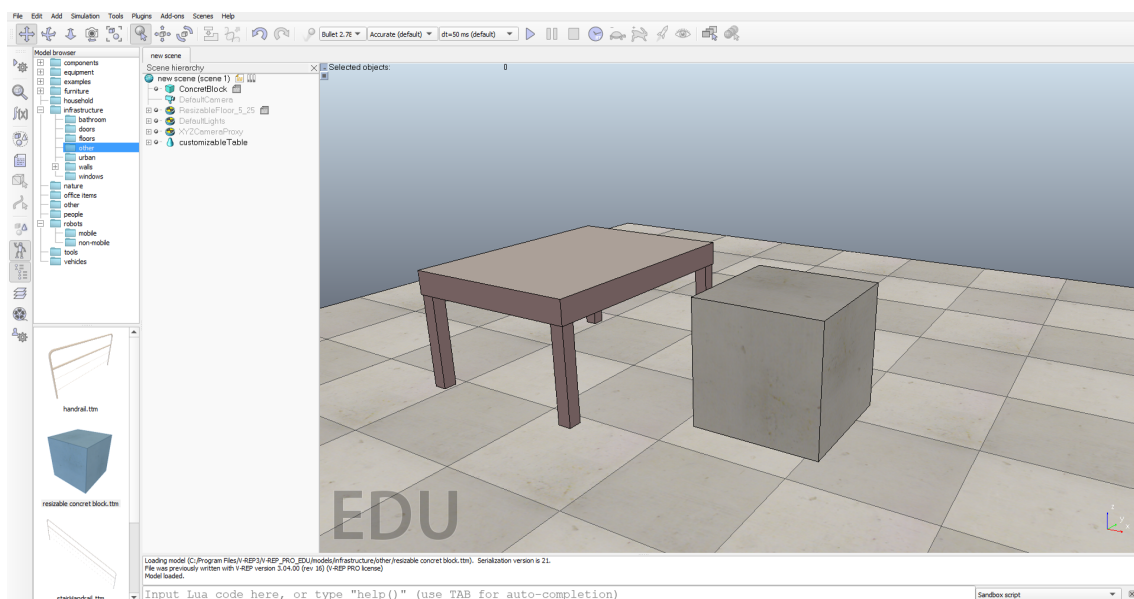


Figura 12.1.2: Passo 2

Neste passo, será adicionado o robô. O robô adicionado será o *KUKA LBR4+*, dentro de *robots > non-mobile*. Selecionando o robô e o bloco de concreto juntos, com a tecla shift, deve-se, novamente, clicar em *Object/Item Shift*, e então pressionar, dentro de *Position* os botões *Apply to selection* para aplicar nos três eixos, assim, o robô se deslocará para dentro do bloco. Em seguida, selecionando apenas o robô e novamente em *Object/Item Shift*, agora em *Translation*, marcando *Own frame* em *Relative to*, deve-se colocar o valor de 0.31 para *Along Z[m]* e clicando em *X-translate sel.*. Após, em *Object/Item Rotation/Orientation*, em *Rotation*, marcando *Own frame* em *Relative to*, deve-se colocar o valor de 90 para *Around Z[deg]* e clicar em *Z-rotate sel.*. Ficará similar a Fig. 12.1.3

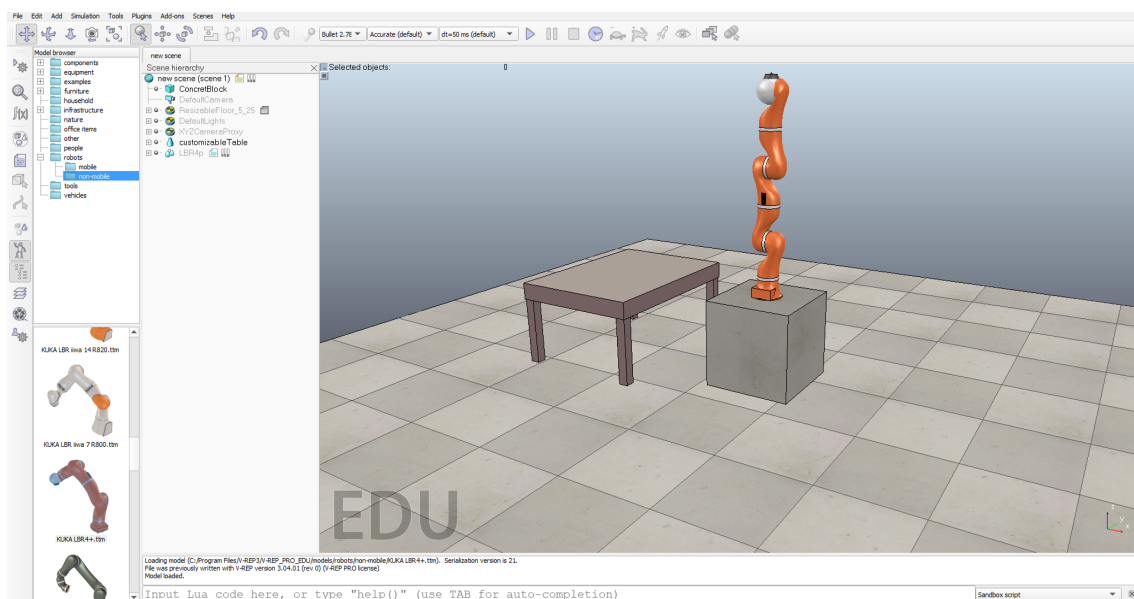


Figura 12.1.3: Passo 3

A hierarquia do cenário, bloco 6 Fig. 10.3.1, é modificada a medida em que objetos são adicionados. Ali, clicando no símbolo de papel escrito ao lado de *LBR4p* há um script embutido no robô que o controla para um movimento pré determinado ao dar play na simulação. Este controle não é interessante para o exemplo, e portanto deve ser apagado.

Para adicionar um copo, será selecionado o objeto *cup* dentro de *households*. Selecionando o copo e a mesa, deve-se clicar em *Object/Item Shift*, pressionar, dentro de *Position* os botões *Apply to selection* para colocar o copo dentro da mesa. Selecionando apenas o copo, deve-se movê-lo, ainda em *Object/Item Shift* em *Translation*, marcando *Own frame* em *Relative to*, deve-se colocar os valores de 0.2, -0.3 e 0.11 para os eixos X, Y e Z, respectivamente, e ajustá-los. Ficará similar à Fig. 12.1.4

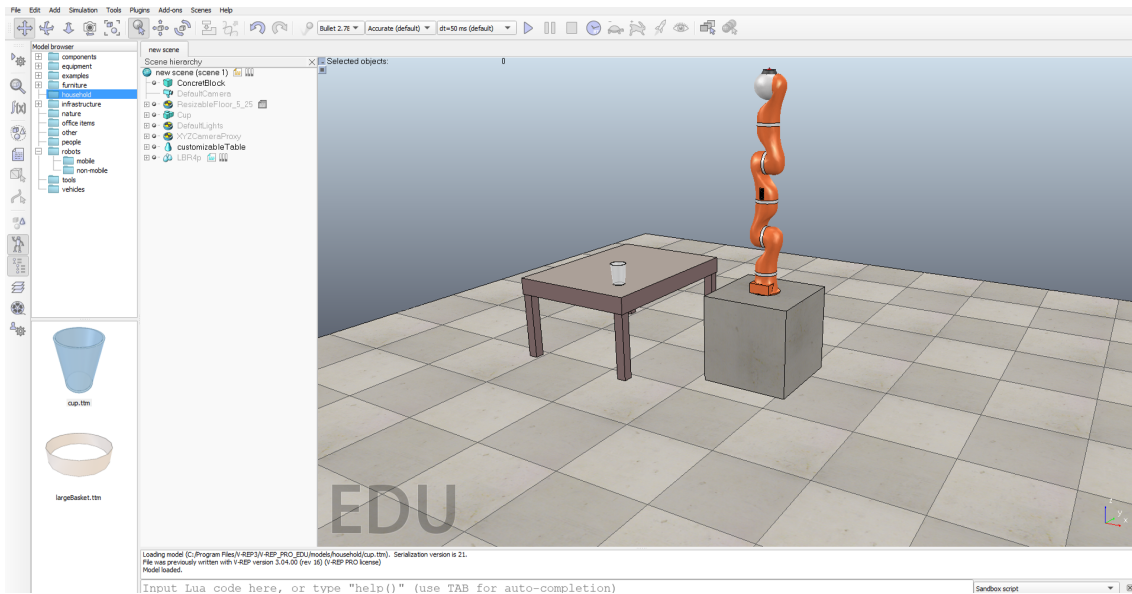


Figura 12.1.4: Passo 4

Por fim, deve-se configurar as juntas do robô e adicionar o efetuador. Para isso, clicando em "+" à esquerda de *LBR4p*, aparecerão os componentes do robô. Em seguida deve-se selecionar as juntas de 1 a 7, *LBR4p_joint1*, *LBR4p_joint2*, ..., até *LBR4p_joint7*. Depois, clicando no bloco 4 Fig. 10.3.1 em *Scene object properties*, na aba *Joints*, deve-se selecionar, em *Mode*, *Torque/Force Mode*. Como mostra a Fig. 12.1.5

Para adicionar o efetuador *BarrettHand*, na biblioteca, em *components > grippers*. Selecionando o efetuador e o componente *LBR4p_connection*, dentro de *LBR4p* na hierarquia da cena, deve-se clicar em *Assemble/Disassemble*, no bloco 2 Fig. 10.3.1. Ficará similar à Fig. 12.1.6

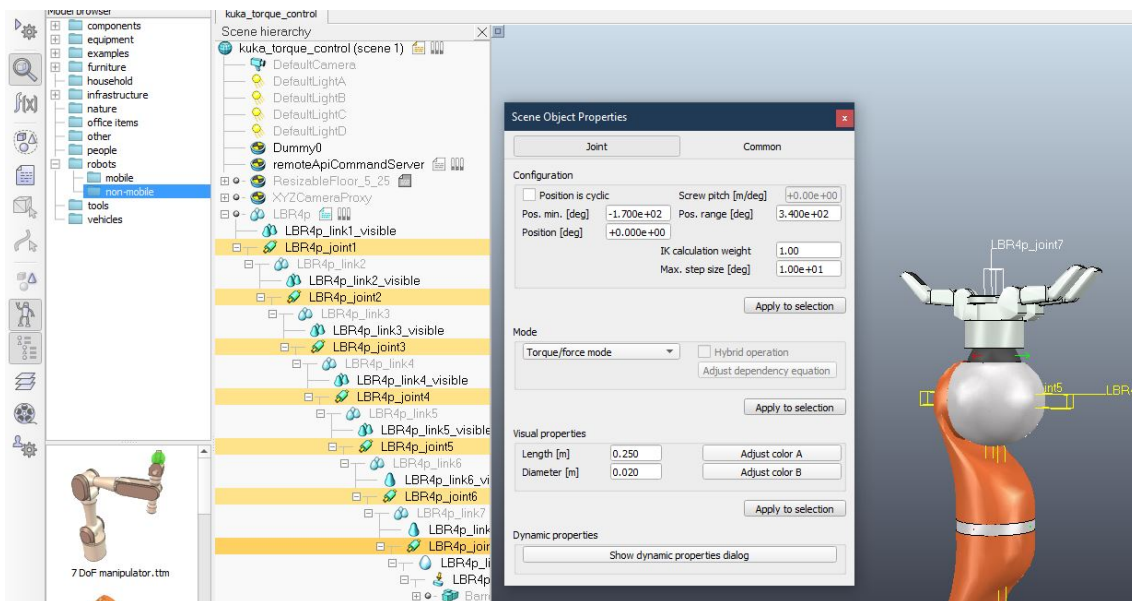


Figura 12.1.5: Passo 5

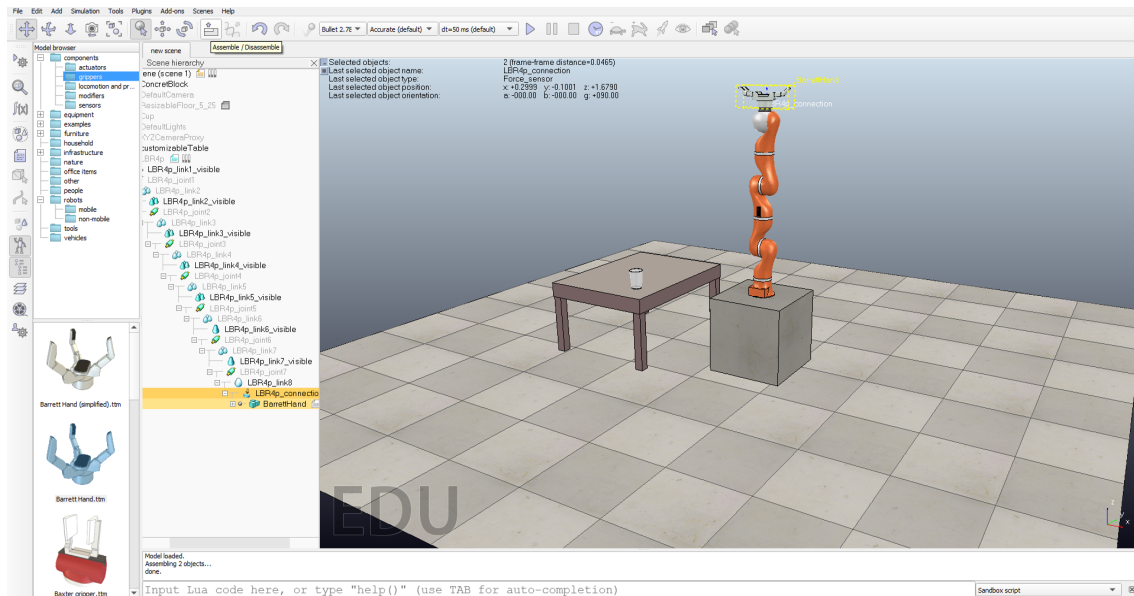


Figura 12.1.6: Passo 6

12.2 Controle Externo

Para facilitar o controle, é recomendada a biblioteca DQRobotics, disponível em:

<https://github.com/dqrobotics>

<https://sourceforge.net/projects/dqrobotics/>

Feito o download da pasta, deve-se adicioná-la ao caminho do Matlab. Para isso, toda vez que for utilizar, deve-se navegar até onde está a pasta no *Current Folder* do Matlab, apertar o botão direito sobre ela, selecionar *Add to Path* e depois *Selected folder and subfolders*. Outra alternativa é adicionar definitivamente a pasta ao caminho do Matlab, assim, não será necessário repetir os passos anteriores toda vez. Para isso, deve-se clicar em *Set Path*, localizado na barra *Home*. Em seguida, clicar em *Add with Subfolders* a pasta desejada.

Com os arquivos *remApi.m*, *remoteApi.dll* e *remoteApiProto.m* no mesmo diretório, um simples comando se dá pelo código a seguir:

kukacontrolsimple.m

```

1      clear ; clc ;
2
3      disp('Program started');
4      vrep=remApi('remoteApi');
5      vrep.simxFinish(-1);
6      clientID=vrep.simxStart('127.0.0.1',19999,true,true
7          ,5000,5);
8
9      [~,j1]=vrep.simxGetObjectHandle(clientID,'LBR4p_joint1',
10         vrep.simx_opmode_oneshot_wait);
11      [~,j2]=vrep.simxGetObjectHandle(clientID,'LBR4p_joint2',

```

```

    , vrep . simx_opmode_one-shot_wait );
10  [~, j3] = vrep . simxGetObjectHandle ( clientID , 'LBR4p_joint3'
    , vrep . simx_opmode_one-shot_wait );
11  [~, j4] = vrep . simxGetObjectHandle ( clientID , 'LBR4p_joint4'
    , vrep . simx_opmode_one-shot_wait );
12  [~, j5] = vrep . simxGetObjectHandle ( clientID , 'LBR4p_joint5'
    , vrep . simx_opmode_one-shot_wait );
13  [~, j6] = vrep . simxGetObjectHandle ( clientID , 'LBR4p_joint6'
    , vrep . simx_opmode_one-shot_wait );
14  [~, j7] = vrep . simxGetObjectHandle ( clientID , 'LBR4p_joint7'
    , vrep . simx_opmode_one-shot_wait );
15
16  joint_handles . j (1) = j1 ;
17  joint_handles . j (2) = j2 ;
18  joint_handles . j (3) = j3 ;
19  joint_handles . j (4) = j4 ;
20  joint_handles . j (5) = j5 ;
21  joint_handles . j (6) = j6 ;
22  joint_handles . j (7) = j7 ;
23
24  startingJoints = [0, pi/3, 0, pi/2, pi/3, 0, 0];
25  disp ( 'Starting robot' );
26
27  vrep . simxPauseCommunication ( clientID , 1 );
28  for i = 1:7
29      vrep . simxSetJointTargetPosition ( clientID ,
30          joint_handles . j (i) , ...
31          startingJoints (i) , ...
32          vrep . simx_opmode_one-shot );
33  end
    vrep . simxPauseCommunication ( clientID , 0 );

```

Em primeiro lugar, é criado um objeto `vrep`, linha 4, que conterà os procedimentos necessários. Também é criada uma variável `clientID` que receberá um número indicando se a conexão foi aberta ou não, linha 6. Esta deverá ter um valor maior que -1. Caso o usuário queira usar outra porta deve alterar o número 19999, desde que também tenha devidamente alterado no cenário. A linha 5 é apenas para fechar caso tenham outras conexões abertas.

Em segundo lugar, deve-se inicializar os handles. Cada junta que se deseja mover deve receber um handle, e a string deve ser exatamente igual ao nome do objeto no Coppeliasim. Como são sete juntas, serão iniciados 7 handles.

Por fim, as juntas são movidas para uma posição especificada pela variável `startingJoints`, linha 24. Para isso, é preciso usar o comando da linha 27 e, para cada handle, dar o argumento desejado. O `for` é usado apenas para simplificar o código, cada iteração é repetida para cada um dos handles. Depois, deve-se usar o comando como o da linha 33 para finalizar o comando.

Assim, o manipulador Kuka é movido para a posição indicada por `startingJoints`.

É deixado para o leitor o desafio de controlar o manipulador e o efetuador para que consiga mover o copo sobre a mesa.

DICA : É também útil usar cinemática direta diferencial para controlar o manipulador. Isto é, fornecer para ele comandos de velocidade ao invés de comandos de posição. Para isso, deve-se voltar ao passo 5, Fig. 12.1.5, clicar em *Show dynamics properties dialog* e desmarcar a caixa de seleção *Control loop enabled*, Fig. 12.2.1.

Feito isso, no código de controle no Matlab deve-se usar o método *simxSetJointTargetVelocity* ao invés do método *simxSetJointTargetPosition* na linha 29.

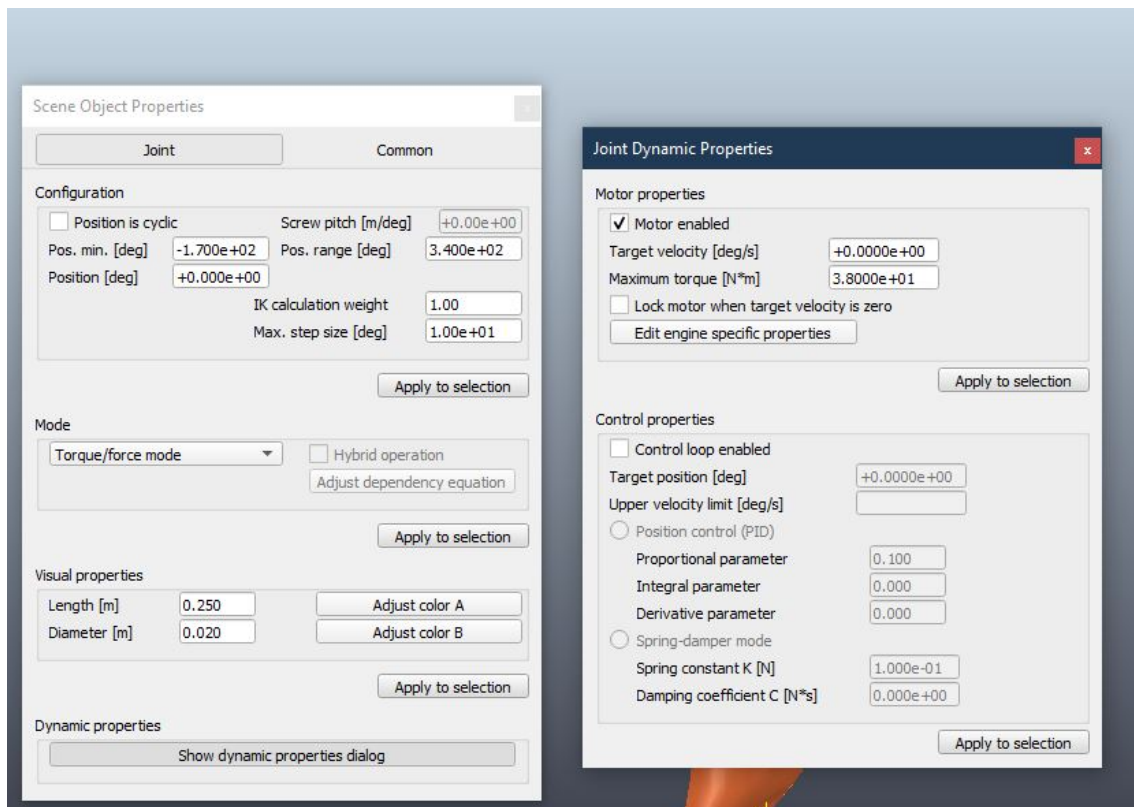
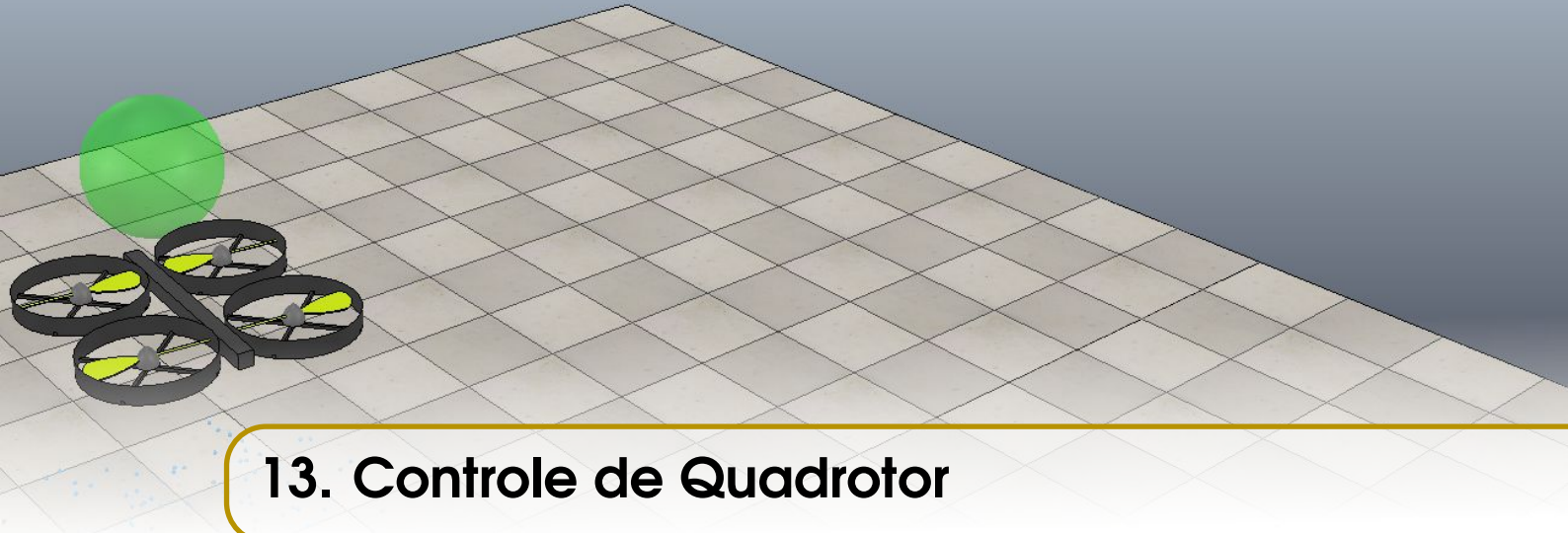
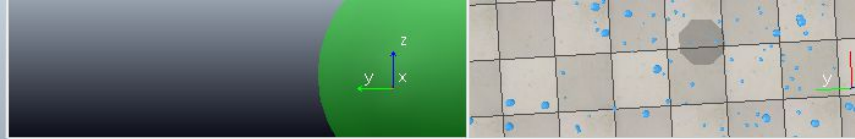


Figura 12.2.1: Usando cinemática direta para controle do manipulador.



13. Controle de Quadrotor

Neste exemplo, um drone no CoppeliaSim será controlado por uma API Remota no Matlab, similar ao exemplo anterior. Para isso, deve-se separar em uma pasta uma cópia dos arquivos: *remApi.m*, *remoteApi.dll* (Se o Sistema Operacional for Windows) e *remoteApiProto.m* para a comunicação remota pelo Matlab e uma cópia do arquivo *remoteApiCommandServerExample.ttt* para o cenário.

Assim como no exemplo anterior, a comunicação será por API Remota Temporária. No objeto *remoteApiCommandServer* há a porta que será utilizada. Por padrão, será a 19999, caso o usuário deseje outra, deverá alterá-la clicando no *Child Script* do objeto.

13.1 Teoria

Denote por $p \in \mathbb{R}^3$ a posição do robô, por $m \in \mathbb{R}$ a sua massa e por $v \in \mathbb{R}^3$ a sua velocidade em relação ao eixo de coordenadas do mundo. Denote também por $R_b^w \in SO(3)$ a matriz de rotação que representa a orientação do robô em relação ao eixo referencial. Adicionalmente, g é a aceleração gravitacional e $\hat{z} = [0, 0, 1]^T$.

$$\dot{p} = v, \quad m\dot{v} = R_b^w \hat{z} \tau - mg \hat{z}, \quad \dot{R}_b^w = R_b^w S(\omega), \quad (13.1.1)$$

em que as entradas são: A força de *thrust* $\tau \in \mathbb{R}$ e as velocidades angulares $\omega \in \mathbb{R}^3$. $S(\omega)$ é a matriz antissimétrica associada ao vetor $\omega = [\omega_x, \omega_y, \omega_z]^T$.

O objetivo é guiar o drone de sua posição p até uma posição alvo p_{goal} . Para isso será calculado um campo vetorial $F(p) : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ que irá mapear uma velocidade de referência desejada. Na verdade, F pode ser $F(p, t) : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}^3$ com o objetivo de mapear velocidades para que o drone siga curvas fechadas variantes no tempo. Será usado

um caso particular desse caso em que F é invariante e tem o objetivo de guiar o drone apenas para uma posição p_{goal} .

Considera-se agora o modelo integrador de segunda ordem dado por:

$$\dot{p} = v, \quad \dot{v} = a_d, \quad (13.1.2)$$

em que a_d é o sinal de entrada. Dado o campo F , a_d pode ser calculado como:

$$a_d = J_F v + k_v(F - v) + \frac{\partial F}{\partial t}. \quad (13.1.3)$$

Ou nesse caso em que F independe de t :

$$a_d = J_F v + k_v(F - v). \quad (13.1.4)$$

Sendo J_F a jacobiana do campo em relação a p e $k_v > 0$ um ganho.

Por fim, é necessário calcular τ e ω .

$$\tau = m(\hat{z}_b)^T a_r, \quad (13.1.5)$$

onde $\hat{z}_b = R_b^w \hat{z}$ é a direção do eixo z do robô e $a_r = a_d + g\hat{z}$ é a aceleração de referência.

Para o cálculo de ω , antes é necessário definir R_r^w e R_e :

$$R_r^w = [\hat{x}_r, \hat{y}_r, \hat{z}_r], \quad \omega_\psi = [\cos(\psi), \sin(\psi), 0]^T, \quad (13.1.6)$$

em que ψ é o *heading angle* que pode ser escolhido livremente,

$$\hat{z}_r = \frac{a_r}{\|a_r\|}, \quad \hat{x}_r = \frac{(\omega_\psi - \omega_\psi^T \hat{z}_b \hat{z}_b)}{\|\omega_\psi - \omega_\psi^T \hat{z}_b \hat{z}_b\|}, \quad \hat{y}_r = \hat{z}_r \times \hat{x}_r \quad (13.1.7)$$

$$R_e = (R_b^w)^T R_r^w. \quad (13.1.8)$$

Em seguida:

$$\omega = \omega_r + k_\beta \sin(\beta) \hat{n}, \quad (13.1.9)$$

onde \hat{n} e β podem ser obtidos pelo Teorema da Rotação de Euler, fazendo um mapeamento $R_e \rightarrow (\hat{n}, \beta)$, em que $trace(R) = R_{11} + R_{22} + R_{33}$:

$$\beta = \cos^{-1}\left(\frac{trace(R_e) - 1}{2}\right), \quad \hat{n} = \frac{1}{2 \sin(\beta)} [R_{e32} - R_{e23}, R_{e13} - R_{e31}, R_{e21} - R_{e12}]^T, \quad (13.1.10)$$

E também onde ω_r pode ser obtido através da relação:

$$S(\omega_r) = (R_b^w)^T \dot{R}_r^w (R_e)^T. \quad (13.1.11)$$

13.2 Cenário

Abrindo a cena, será adicionado um modelo de drone built-in no simulador. Ele pode ser localizado no *Model browser*, dentro de robots, dentro de mobile. Nessa pasta, deve-se localizar o modelo *quadricopter.ttm*, Fig. 13.2.1.

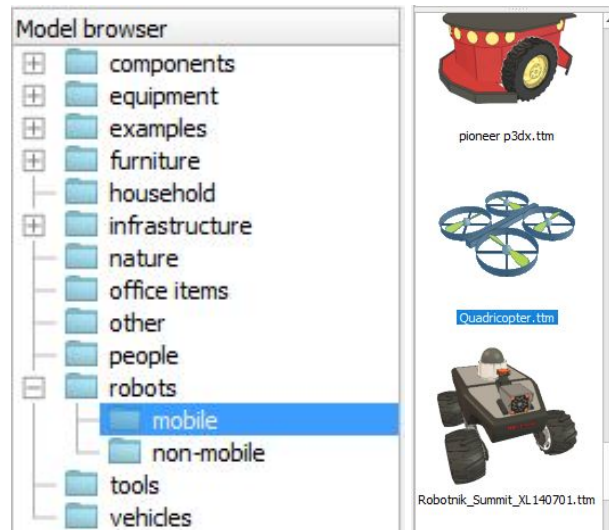


Figura 13.2.1: Localizando o modelo *quadricopter.ttm*.

Depois de adicionado o drone, a hierarquia da cena deverá se parecer com a Fig. 13.2.2.

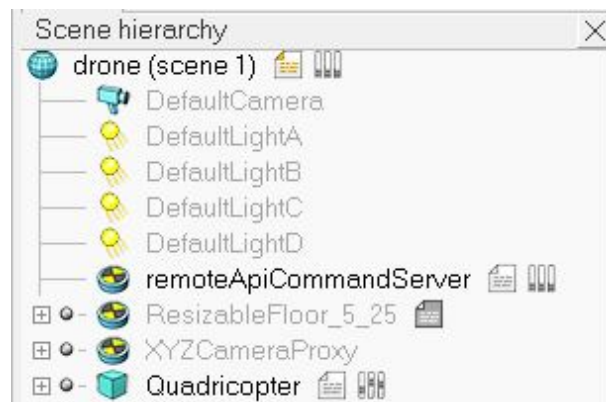


Figura 13.2.2: Hierarquia da cena após a adição do drone.

Esse modelo já possui um controlador. Ao dar play na simulação e transladar o objeto *Quatricopter_target*, uma esfera verde, com a ferramenta de translação na barra superior (*Object/item shift*) o drone seguirá esse objeto. Mas o objetivo aqui é implementar um novo controle pelo Matlab.

Desta forma, o Child Script do modelo será alterado a fim de estabelecer apenas um controle de baixo nível e de criar uma função que será chamada pelo Matlab. Para abri-lo deve-se clicar no símbolo de papel ao lado do nome do objeto *Quadricopter*.

Em primeiro lugar, deve-se adicionar ao fim da função de inicialização

```
function sysCall_init()
```

as seguintes variáveis:

```

exi = 0
eyi = 0
ezi = 0

angular_rate = {0.0, 0.0, 0.0}
thrust_input = 5.335

```

Em seguida, deve-se modificar a função de atuação (*sysCall_actuation*) para remover o controle existente e adicionar apenas um controle PI de baixo nível. Ela deverá ficar parecida com:

```

function sysCall_actuation()
    -- Fake shadow
    s=sim.getObjectSizeFactor(d)
    pos=sim.getObjectPosition(d,-1)
    if (fakeShadow) then
        itemData={pos[1],pos[2],0.002,0,0,1,0.2*s}
        sim.addDrawingObjectItem(shadowCont,itemData)
    end

    -- Acro mode control
    lin_vel, ang_vel =sim.getObjectVelocity(d)
    m = sim.getObjectMatrix(d,-1)
    m_inv = simGetInvertedMatrix(m)
    M = m_inv
    body_vel = {0,0,0}
    body_vel[1] = M[1]*ang_vel[1] + M[2]*ang_vel[2] + M
        [3]*ang_vel[3]
    body_vel[2] = M[5]*ang_vel[1] + M[6]*ang_vel[2] + M
        [7]*ang_vel[3]
    body_vel[3] = M[9]*ang_vel[1] + M[10]*ang_vel[2] + M
        [11]*ang_vel[3]

    Kp = 0.35;
    Ki = 0.00;

    tau_r = thrust_input -- reference for relative thrust
        , in relation to the hover value
    wr = angular_rate -- reference for the angular
        velocities
    ex = wr[1]-body_vel[1] -- angular velocity error on x
    ey = wr[2]-body_vel[2] -- angular velocity error on y
    ez = wr[3]-body_vel[3] -- angular velocity error on z
    dt = sim.getSimulationTimeStep() -- simulation step
    exi = exi + ex*dt -- integral of ex
    eyi = eyi + ey*dt -- integral of ey
    ezi = ezi + ez*dt -- integral of ez

```

```

cx = Kp*ex + Ki*exi -- control law for x momentum
cy = Kp*ey + Ki*eyi -- control law for y momentum
cz = Kp*ez + Ki*ezi -- control law for z momentum

-- Decide of the motor velocities:
particlesTargetVelocities [1]=tau_r*(1.0) +cx -cy -cz
particlesTargetVelocities [2]=tau_r*(1.0) +cx +cy +cz
particlesTargetVelocities [3]=tau_r*(1.0) -cx +cy -cz
particlesTargetVelocities [4]=tau_r*(1.0) -cx -cy +cz

-- Send the desired motor velocities to the 4 rotors:
for i=1,4,1 do
    sim.setScriptSimulationParameter(propellerScripts
        [i], 'particleVelocity',
        particlesTargetVelocities [i])
end

end

```

Por último, deve-se criar uma nova função que será utilizada pelo Matlab.

```

setSpeed = function(inInts, inFloats, inStrings, inBuffer
)
    thrust_input = inFloats [1]
    angular_rate = {inFloats [2], inFloats [3], inFloats
        [4]}
    return {}, {}, {}, ''
end

```

Depois disso, o cenário está devidamente configurado e o drone receberá do Matlab as entradas τ e ω no formato $[\tau, \omega_x, \omega_y, \omega_z]$.

13.3 Implementação

Nesta última seção, serão abordadas as implementações do controle no Matlab. Os arquivos descritos aqui devem estar na mesma pasta dos arquivos necessários para a criação de uma API Remota, citados no início deste exemplo.

A seguir estão as cinco funções auxiliares criadas e o script principal.

axisangle_from_R.m

Esta função retorna o eixo e o ângulo a partir de uma matriz de rotação.

```

1 function [n, B] = axisangle_from_R(R)
2     % [n, B] = axisangle_from_R(R)
3     % returns the axis/angle representation using Euler's
    rotation theorem

```

```

4     % when given a rotation matrix as input
5     B = acos((trace(R)-1)/2);
6     n = (1/(2*sin(B)))* [R(3,2)-R(2,3); R(1,3)-R(3,1);R
      (2,1)-R(1,2)];
7     if B == 0 % When B == 0, n is undefined
8         n = [0;0;0]
9     end
10 end

```

double_integrator.m

Esta função calcula a entrada a_d do modelo integrador de segunda ordem.

```

1 function ad = double_integrator(F,v,p,kv,t)
2     % ad = double_integrator(F,v,p,kv)
3     % returns the second integrator model input ad
4     % Inputs:
5     % F is the Vector Field struct
6     % v is the robot velocity
7     % p is the robot position
8     % kv > 0 is a gain
9     % t is the simulation time
10    JF = [F.dFdx(p,t) F.dFdy(p,t) F.dFdz(p,t)];
11    ad = JF*v + kv*(F.F(p,t) - v) + F.dFdt(p,t);
12 end

```

generateFieldEquations

Esta função gera a *struct* referente ao campo vetorial para ser usado na função anterior.

```

1 function F = generateFieldEquations(type,p_goal)
2     % F = generateFieldEquations(type,p_goal)
3     % returns an attractive vector field to p_goal. F will
      be a struct with
4     % F.F: the field function handle
5     % F.dFdx: the x partial derivative function handle
6     % F.dFdy: the y partial derivative function handle
7     % F.dFdz: the z partial derivative function handle
8     % F.dFdt: the t partial derivative function handle
9     d = 1; c = 1;
10    switch type
11        case 1
12            F.F = @(p,t) (norm(p-p_goal)<=d).*(-c*(p-p_goal)
      ) + ...
13            (norm(p-p_goal)>d).*(-d*c*(p-p_goal)/(norm(p-
      p_goal)+1e-6));
14
15            F.dFdx = @(p,t) [-c;0;0];
16            F.dFdy = @(p,t) [0;-c;0];
17            F.dFdz = @(p,t) [0;0;-c];

```

```

18         F.dFdt = @(p,t) [0;0;0];
19     case 2
20         A = [-1 0.1 0; -0.1 -1 0; 0 0 -1];
21         F.F = @(p,t) (c*A*(p-p_goal));
22         F.dFdx = @(p,t) c*[-1;-0.1;0];
23         F.dFdy = @(p,t) c*[0.1;-1;0];
24         F.dFdz = @(p,t) c*[0;0;-1];
25         F.dFdt = @(p,t) c*[0;0;0];
26     case 3
27         A = [-1 0.4 0; -0.4 -1 0; 0 0 -1];
28         F.F = @(p,t) (c*A*(p-p_goal));
29         F.dFdx = @(p,t) c*[-1;-0.4;0];
30         F.dFdy = @(p,t) c*[0.4;-1;0];
31         F.dFdz = @(p,t) c*[0;0;-1];
32         F.dFdt = @(p,t) c*[0;0;0];
33 end

```

generateRwr.m

Esta função gera a matriz de rotação R_r^w .

```

1 function Rwr = generateRwr(ar, zb)
2     % Rwr = generateRwr(ar, zb)
3     % generates de Rwr rotation matrix
4     psi = 0;
5     wpsi = [cos(psi) sin(psi) 0]';
6     zr = ar/norm(ar);
7     xr = (wpsi - wpsi'*zb*zb); xr = xr/norm(xr);
8     yr = cross(zr, xr);
9     Rwr = [xr yr zr];
10 end

```

R_from_euler.m

Essa função retorna a matriz de rotação a partir da orientação do robô.

```

1 function R = R_from_euler(euler)
2     % R = R_from_euler(euler)
3     % generates a Rzyz rotation matrix from euler angles
4     representation
5     phi = euler(1);
6     theta = euler(2);
7     psi = euler(3);
8     Rz1 = [cos(phi) -sin(phi) 0
9            sin(phi)  cos(phi) 0
10           0          0      1];
11     Ry = [cos(theta) 0 sin(theta)
12           0          1      0
13           -sin(theta) 0 cos(theta)];
14     Rz2 = [cos(psi) -sin(psi) 0

```

```

14             sin(psi)  cos(psi)  0
15             0         0         1];
16     R = Rz1 * Ry * Rz2;
17 end

```

main.m

Finalmente, o código principal referente ao controle está abaixo. Nele, pode-se ressaltar as seguintes características:

Nas linhas 14 até 18 estão as entradas do usuário. A primeira, *targets*, é uma lista de pontos alvos que o drone deverá seguir, enquanto que a segunda, *iterations*, é o número de iterações usadas para cada perseguição do ponto alvo.

Nas linhas 22 até 25 estão as inicializações da API. Da linha 32 até a 45 estão algumas variáveis que serão utilizadas pelo programa.

A partir da linha 48 a simulação é iniciada com dois laços *for* aninhados. O primeiro e mais externo é responsável por percorrer a lista de pontos alvo. Nele, o objeto alvo é transladado até a posição desejada e o campo *F* é calculado. Já o segundo laço, e mais interno, é responsável por guiar o drone até o ponto alvo *i*. Dentro dele o controle é executado e os sinais τ e ω são enviados para o CoppeliaSim na linha 76, através da função *setSpeed* que definimos no script em Lua na primeira seção.

Depois disso, das linhas 81 até 85 a simulação e a comunicação são finalizadas.

```

1 %% MAIN
2 % Code fs pelo programa. E a partir da or drone control
   with remote Api communication with CoppeliaSim
3 %
4 % Arthur Nunes
5 %
6 % Need auxiliar functions: axisangle_from_R ,
   double_integrator ,
7 % generateFieldEquations , generateRwr and R_from_euler
8 % Need remote Api files: remApi.m, remoteApi.dll (Windows) ,
9 % remoteApiProto.m
10 clear , clc , close all
11
12 %% User inputs
13
14 targets = [-1 0 3.5
15           0 0 3.5
16           0 0 2
17           0.2 -0.5 3];
18 iterations = 180;
19
20 %% API Initialization
21 % First play the simulation scene

```

```

22 vrep = remApi('remoteApi');
23 vrep.simxFinish(-1);
24 clientID=vrep.simxStart('127.0.0.1', 19999, true, true,
    5000, 5);
25 vrep.simxSynchronous(clientID, true);
26 vrep.simxStartSimulation(clientID, vrep.simx_opmode_oneshot
    );
27
28 [~, target_handle] = vrep.simxGetObjectHandle(clientID, '
    Quadricopter_target', vrep.simx_opmode_oneshot_wait);
29 [~, drone_handle] = vrep.simxGetObjectHandle(clientID, '
    Quadricopter_base', vrep.simx_opmode_oneshot_wait);
30
31 %% Variables
32 [~, target_pos] = vrep.simxGetObjectPosition(clientID,
    target_handle, -1, vrep.simx_opmode_oneshot_wait);
33
34 [~, drone_pos] = vrep.simxGetObjectPosition(clientID,
    drone_handle, -1, vrep.simx_opmode_oneshot_wait);
35 [~, drone_orient] = vrep.simxGetObjectOrientation(clientID,
    drone_handle, -1, vrep.simx_opmode_oneshot_wait);
36 %[~, drone_quat] = vrep.simxGetObjectQuaternion(clientID,
    drone_handle, -1, vrep.simx_opmode_oneshot_wait);
37 [~, drone_linvel, drone_angvel] = vrep.
    simxGetObjectVelocity(clientID, drone_handle, vrep.
    simx_opmode_oneshot_wait);
38
39 inputs = [5.335, 0, 0, 0];
40 g = 9.81;
41 m = inputs(1)/g;
42 kbeta = 0.15;
43 kv = 0.15;
44
45 prev_Rwr = eye(3);
46
47 %% Simulation
48 for i = 1:size(targets,1)
49     new_target_pos = targets(i,:);
50     vrep.simxSetObjectPosition(clientID, target_handle, -1,
        new_target_pos, vrep.simx_opmode_oneshot_wait);
51     [~, target_pos] = vrep.simxGetObjectPosition(clientID,
        target_handle, -1, vrep.simx_opmode_oneshot_wait);
52     F = generateFieldEquations(1, target_pos');
53
54     for t = 1:iterations
55         %[~, target_pos] = vrep.simxGetObjectPosition(
            clientID, target_handle, -1, vrep.
            simx_opmode_oneshot_wait);

```

```

56     [~, drone_pos] = vrep.simxGetObjectPosition(
        clientID, drone_handle, -1, vrep.
        simx_opmode_oneshot_wait);
57     [~, drone_orient] = vrep.simxGetObjectOrientation(
        clientID, drone_handle, -1, vrep.
        simx_opmode_oneshot_wait);
58     % [~, drone_quat] = vrep.simxGetObjectQuaternion(
        clientID, drone_handle, -1, vrep.
        simx_opmode_oneshot_wait);
59     [~, drone_linvel, drone_angvel] = vrep.
        simxGetObjectVelocity(clientID, drone_handle,
        vrep.simx_opmode_oneshot_wait);

60
61     ad = double_integrator(F, drone_linvel', drone_pos',
        kv, -1);
62     Rwb = R_from_euler(drone_orient');
63     zb = Rwb*[0 0 1]';
64     ar = ad + g*[0 0 1]';
65     Rwr = generateRwr(ar, zb);
66     Re = Rwb'*Rwr;
67     [n, Beta] = axisangle_from_R(Re);
68
69     Rwr_dot = Rwr - prev_Rwr; prev_Rwr = Rwr;
70     S_wr=(Rwb')*Rwr_dot*(Re');
71     wr = [S_wr(3,2), S_wr(1,3), S_wr(1,2)]';
72
73     tau = m*zb'*ar;
74     w = wr +kbeta*sin(Beta)*n;
75     inputs = [tau, w'];
76     vrep.simxCallScriptFunction(clientID, 'Quadricopter'
        , vrep.sim_scripttype_childscript, 'setSpeed'
        ,[], inputs, [], [], vrep.simx_opmode_blocking);
77     end
78 end
79
80 %% End the API and the simulation
81 vrep.simxPauseSimulation(clientID, vrep.simx_opmode_oneshot)
    ;
82 vrep.simxStopSimulation(clientID, vrep.
    simx_opmode_oneshot_wait);
83 vrep.simxFinish(clientID);
84 vrep.delete();
85 disp('Program ended');

```

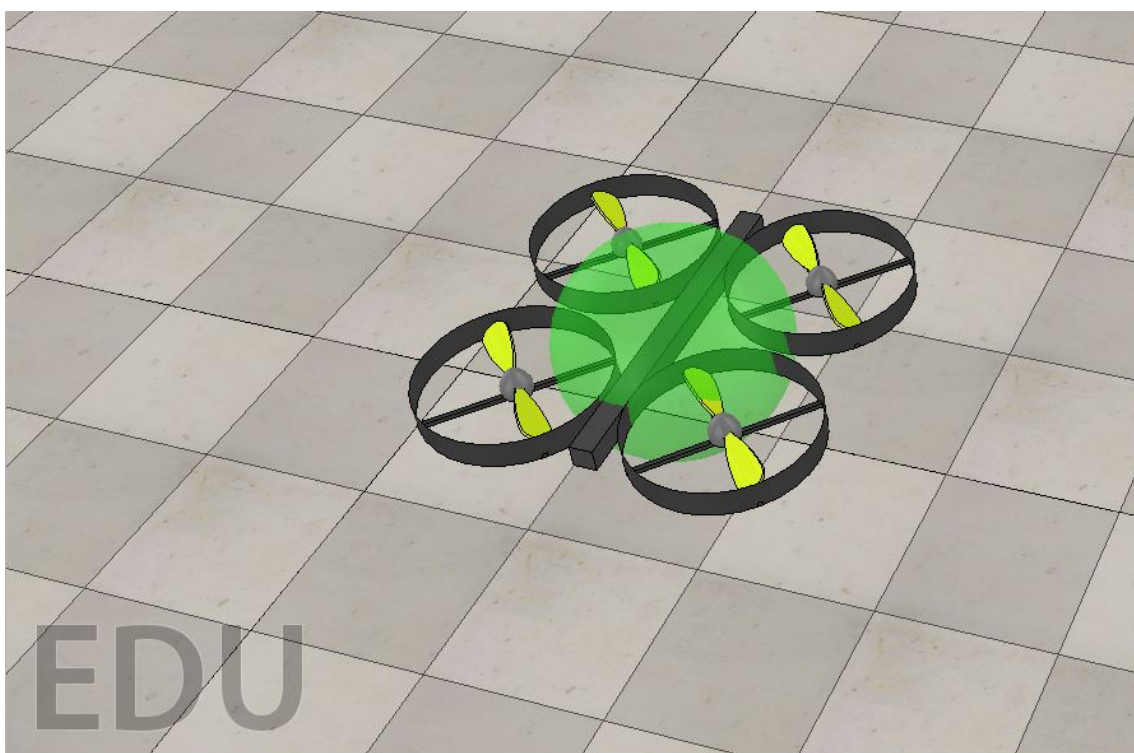
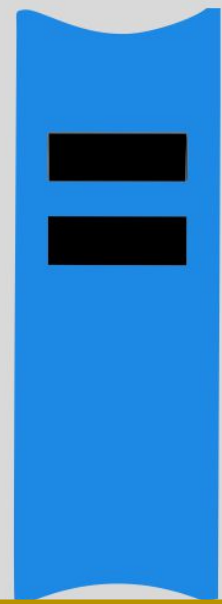
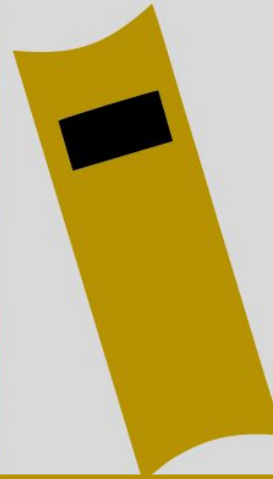



Figura 13.3.1: *Simulação com o quadrotor.*

IV

Índice

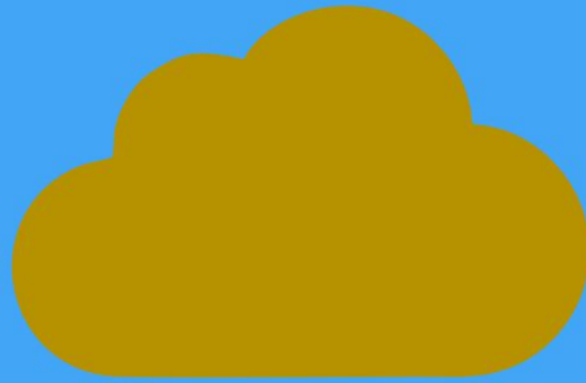
Referências Bibliográficas	141
Repositórios	143
Apêndice I - AAI Robotics	145
Apêndice II - CORO	147
Instalação	
Joystick	
iRobot Create	
Câmeras	



Referências Bibliográficas

- [1] H. M. Choset, S. Hutchinson, K. M. Lynch, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.
- [2] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.
- [3] M. W. Spong, S. Hutchinson, M. Vidyasagar, *et al.*, *Robot modeling and control*. 2006.
- [4] A. H. Nunes, I. F. Amaral, and PETEE-UFMG, *Learning.py: Uma apostila de introdução à programação em Python*. 2020. Disponível em www.petee.cpdee.ufmg.br/.
- [5] A. H. Nunes and PETEE-UFMG, *Learning.mat: Uma apostila de introdução às ferramentas do Matlab*. 2020. Disponível em www.petee.cpdee.ufmg.br/.
- [6] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: a practical introduction to the Robot Operating System*. "O'Reilly Media, Inc.", 2015.
- [7] A. M. Rezende, V. M. Gonçalves, A. H. Nunes, and L. C. Pimenta, "Robust quadcopter control with artificial vector fields," in *2020 IEEE/RSJ International Conference on Robotics and Automation (ICRA)*, IEEE, 2020 (Accepted for publication).
- [8] G. Garcia, F. Rocha, M. Torre, W. Serrantola, F. Lizarralde, A. Franca, G. Pessin, and G. Freitas, "ROSI: A Novel Robotic Method for Belt Conveyor Structures Inspection," in *2019 19th International Conference on Advanced Robotics (ICAR)*, pp. 326–331, Dec 2019.
- [9] D. Bore, A. Rana, N. Kolhare, and U. Shinde, "Automated Guided Vehicle Using Robot Operating Systems," in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 819–822, IEEE, 2019.

- [10] S. Ergur and M. Ozkan, “Trajectory planning of industrial robots for 3-D visualization a ROS-based simulation framework,” in *2014 IEEE International Symposium on Robotics and Manufacturing Automation (ROMA)*, pp. 206–211, IEEE, 2014.
- [11] B. Mirkhanzadeh, C. Shao, A. Shakeri, T. Sato, M. Razo-Razo, M. Tacca, A. Fumagalli, and N. Yamanaka, “A two-layer network Orchestrator offering trustworthy connectivity to a ROS-industrial application,” in *2017 19th International Conference on Transparent Optical Networks (ICTON)*, pp. 1–4, IEEE, 2017.
- [12] “Tutoriais sobre ROS (Oficial).” wiki.ros.org/pt_BR/ROS/Tutorials. Acessado em Dezembro 2019.
- [13] “Next generation robots: Autonomous subsurface explorers.” siamagazin.com/next-generation-robots-autonomous-subsurface-explorers/. Acessado em Abril 2020.
- [14] “Stack overflow.” stackoverflow.com/. Acessado em Abril 2020.
- [15] “Mundo ubuntu.” mundoubuntu.com.br/. Acessado em Dezembro 2019.
- [16] “Viva o linux.” vivaolinux.com.br/. Acessado em Dezembro 2019.



Repositórios

Apostila The Turtles

<https://github.com/ArthurHDN/ApostilaTheTurtles>

AAI Robotics - ROSI Challenge

https://github.com/ara1557/aai_robotics

ROSI Challenge

<https://github.com/filRocha/rosiChallenge-sbai2019>

Modelo do iRobot para o Gazebo

https://github.com/joelillo/create_simulator/tree/master/model

Gazebo tutorials

https://bitbucket.org/osrf/gazebo_tutorials/src

Turtlebot para ROS Kinetic

https://github.com/turtlebot/turtlebot_create

DQ Robotics para Matlab

<https://github.com/dqrobotics>

AAI Robotics

Apêndice I - AAI Robotics

Esta apostila é dedicada a equipe AAI Robotics que venceu em terceiro lugar o desafio de programação de robôs autônomos ROSI Challenge, no 14º Simpósio de Automação Inteligente, em Ouro Preto, MG. A equipe foi em nome do grupo PETEE, representando a Universidade Federal de Minas Gerais.

O repositório das soluções encontradas pela equipe para o problema de inspeção de transportadoras de correias está disponível em https://github.com/ara1557/aai_robotics

O simulador desenvolvido pela comissão organizadora está disponível em <https://github.com/filRocha/rosiChallenge-sbai2019>

Foi uma ótima experiência, além de emocionante. Os sinceros agradecimentos a equipe do grupo PETEE por participarem conosco e por nos apoiarem. À UFMG, à Vale e ao Instituto Tecnológico Vale (ITV) pela oportunidade de participar do SBAI, de participar da competição e de aprendizado e crescimento.



Figura 13.3.2: Equipe AAI Robotics



Figura 13.3.3: Equipe PETEE UFMG no 14º SBAI



Apêndice II - CORO

Instalação

Para o laboratório deve-se instalar o Linux da distribuição Ubuntu, versão 16.04 LTS. Guia completo para o dual boot na página 11.

- Antes da instalação, deve-se desativar a inicialização rápida do Windows, o que poderá dificultar a fácil troca de sistema operacional. Para isso vá em "Painel de Controle", "Opções de Energia", "Escolher a função dos botões de energia", "Alterar configurações não disponíveis no momento", e desmarque a opção "Ligar inicialização rápida (recomendado)" e clique em "Salvar alterações". Agora poderá prosseguir com a instalação:
 - 1 É necessário baixar o arquivo ISO do Ubuntu 16.04 LTS, que está disponível gratuitamente em <https://www.ubuntu.com/download/desktop>
 - 2 Crie um pendrive bootável com o Ubuntu.
 - 3 Abra o gerenciador de disco. Para isso pode ser apertar "Windows"+ "R", digite "diskmgmt.msc" e pressionar "Enter" ou aperte com o botão direito em Este Computador, Gerenciar e selecionar Gerenciador de Disco
 - 4 Clique com o botão direito na unidade que deseja particionar e selecione "Diminuir Volume". Recomendamos uma partição de 100Gb = 102400Mb. Após, deve aparecer um espaço de 100Gb - ou do tamanho liberado pelo usuário - não alocado. É onde será instalado o Ubuntu.
 - 5 Desligue a máquina, plugue o pendrive e ligue a máquina. Abra a BIOS de seu computador. Cada marca possui um atalho diferente, mas geralmente é pressionar algumas das teclas "Delete", "F12", "F10", "Esc", enquanto o computador inicia.
 - 6 Selecione a língua que deseja e em seguida, em Instalar Ubuntu.
 - 7 Selecione a partição que irá receber o Ubuntu e clique em "+" para adicionar nova partição.

- 8 No tamanho, recomendamos que seja igual ou o dobro da quantidade de memória RAM de sua máquina. Tipo: "Lógica"; Localização para nova partição: "Início deste espaço"; Usar como: "Área de troca (swap)".
- 9 Novamente, seleciona o espaço que sobrou e clique em "+" para adicionar nova partição.
- 10 Tamanho deixe o resto que sobrou. Tipo: "Lógica"; Localização para nova partição: "Início deste espaço"; Usar como: "Sistema de arquivos com "journaling; Ponto de montagem: "/".
- 11 Selecione a última partição realizada e prossiga com a instalação.

Em seguida, instalar o ROS Kinetic Kame, com seu guia completo na página 21.

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

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

$ sudo apt-get update

$ sudo apt-get install ros-kinetic-desktop-full

$ sudo rosdep init

$ rosdep update

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

$ source ~/.bashrc

$ sudo apt install python-rosinstall python-rosinstall-
generator python-wstool build-essential
```

Joystick

Para conectar o Joystick do laboratório, é necessário o pacote *joy*, que pode ser instalado via `sudo apt-get install ros-kinetic-joy` ou encontrado no repositório: https://github.com/ros-drivers/joystick_drivers.

Antes de executar o ROS e o nó, é necessário verificar se o Joystick está funcionando. Para isso, é preciso verificar se o Linux está reconhecendo o Joystick. Plugue-o na máquina e execute o comando `ls /dev/input`. Deverá aparecer algum nome como `js<X>`, X geralmente é 0. Após isso, teste o Joystick com o comando `sudo jstest /dev/input/js<X>`. No terminal

os botões e eixos do Joystick aparecerão e deverão mudar enquanto são alterados. Realize os testes.

Feito isso, basta garantir permissões ao Joystick para que o ROS consiga usá-lo:

```
$ chmod 777 /dev/input/js<X>
```

O nó que lê o joystick é o nó, em C++, *joy_node*, que lê as informações do Joystick e publica em um tópico *joy*.

Para transferir os dados de *joy* para um *cmd_vel*, por exemplo, é preciso fazer outro nó, como:

teleop_joy.py

```
1 #!/usr/bin/env python
2 import rospy
3 from geometry_msgs.msg import Twist
4 from sensor_msgs.msg import Joy
5
6 def callback(data):
7     global Kp, vel_pub
8     Kp = 0.1
9     vel_pub = Twist()
10    vel_pub.linear.x = Kp*data.axes[1]
11    vel_pub.angular.z = Kp*data.axes[0]
12    pub.publish(vel_pub)
13
14 def start():
15     global pub
16     pub = rospy.Publisher('/cmd_vel', Twist, queue_size =
17         1)
18     rospy.Subscriber("joy", Joy, callback)
19     rospy.init_node('joy2robot', anonymous=True)
20     rospy.spin()
21
22 if __name__ == '__main__':
23     try:
24         start()
25     except rospy.ROSInterruptException:
26         pass
```

iRobot Create

Há quatro modelos de robôs iRobot Create, como mostra a Fig 13.3.4 e quatro computadores netbooks brancos no laboratório.

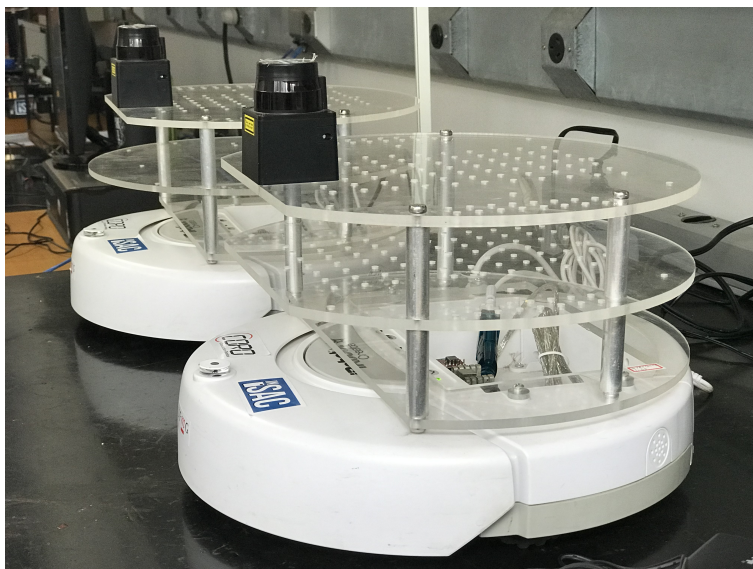


Figura 13.3.4: Modelos iRobot Create do Laboratório

Para conectar-se a um robô deve: Verificar se os robôs estão carregados e, se possível, carregá-los depois do uso.

- Iniciar algum netbook. O login e a senha são **coro, coro**.
- Plugar a saída USB do robô no computador.
- Abrir um terminal.
- Executar *roscore*.
- Abrir um novo terminal.
- Executar *rosvun turtlebot turtlebot_node.py*

Para conectar-se usando outros computadores ao mesmo robô, é necessário o drive *turtlebot* que contém o nó *turtlebot_node.py*, responsável por iniciar o robô. Os pacotes podem ser encontrados neste repositório:

https://github.com/turtlebot/turtlebot_create Talvez seja necessário calibrar a odometria.

Câmeras no Teto

Para conectar as câmeras do teto é preciso ligar o computador que recebe o USB - login e senha são **coro, coro**-, abrir o terminal e executar *fiducialrun*.

Feito isso, em um dos computadores branquinhos deve-se executar *rosvun fiducialros fiducialros*. Os computadores precisam estar ligados na mesma rede.

Isso já bastará para que o branquinho reconheça o marcador. Cada computador está configurado para ler um marcador em específico. São quatro marcadores, que estão impressos e pregados em cada um dos branquinhos.

Caso dê erro no *sock 3* deve-se desligar tudo e tentar novamente.