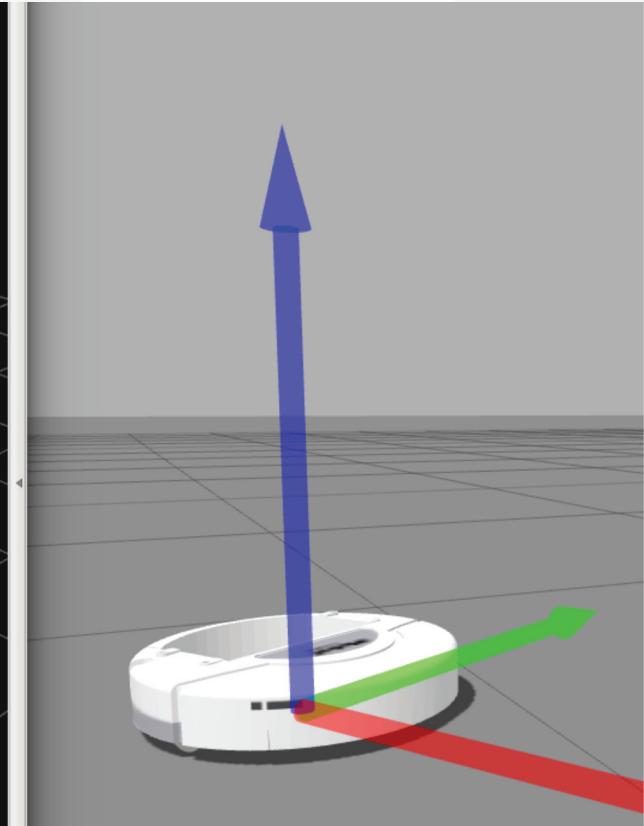
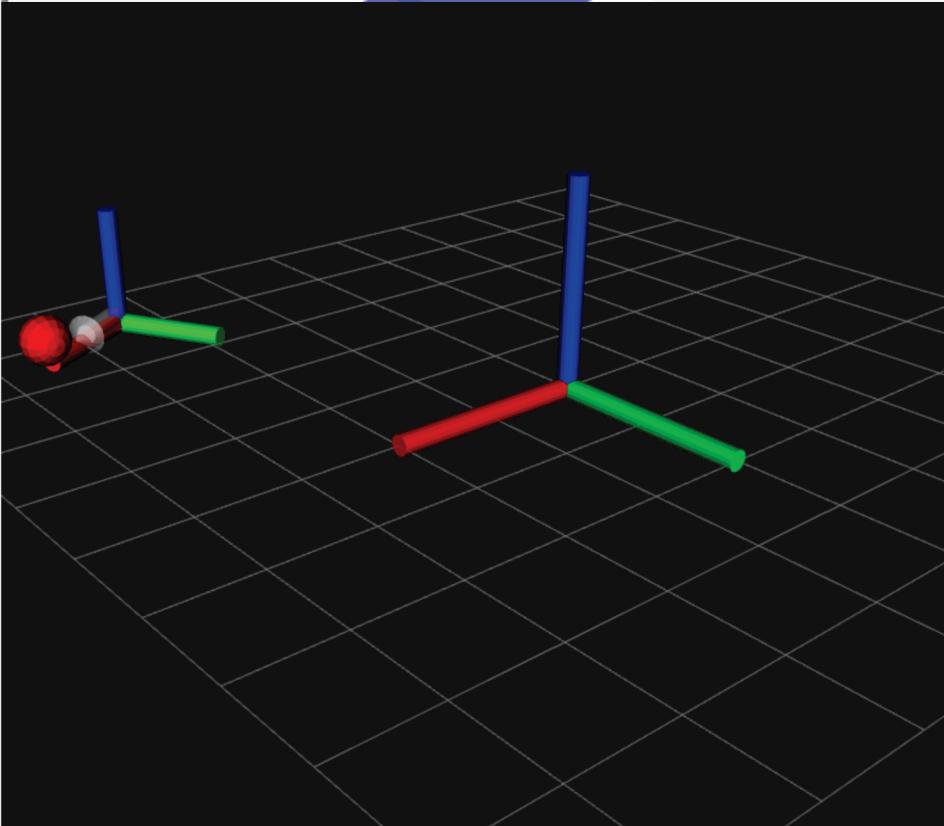
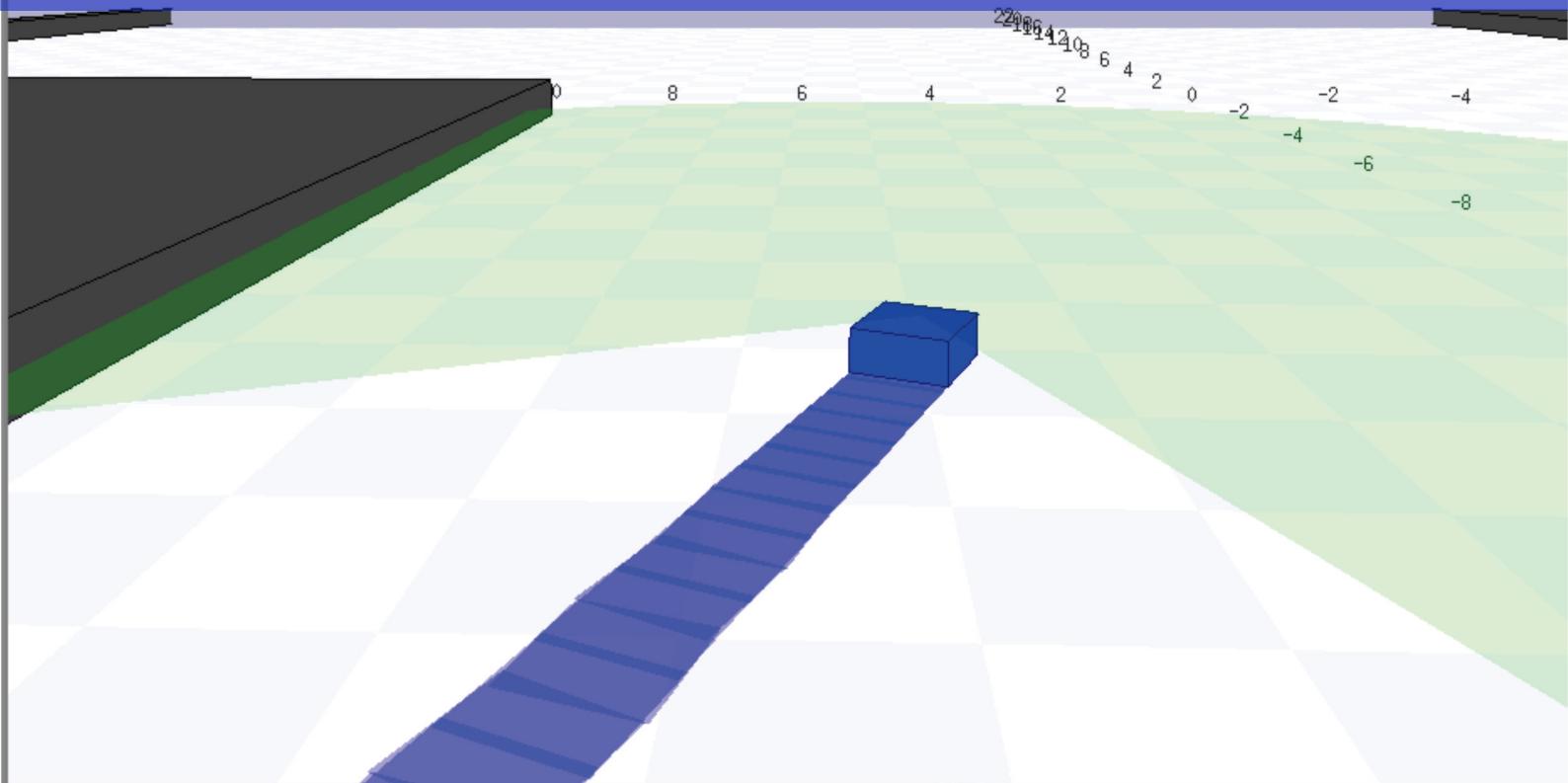


Jornada ROS

O Guia Prático de Robot Operating System

Arthur Henrique Dias Nunes



Prefácio

O Robot Operating System (ROS) é uma ferramenta crucial em nosso laboratório CORO para desenvolvermos pesquisas. Utilizamos o ROS Kinetic Kame. Há a necessidade de guiar novos integrantes, bem como alunos, no início da utilização do software e as vezes, no uso do Linux. Diante dessa necessidade, foi feita esta apostila.

Esta apostila mostra o básico para que possam desenvolver e tratar sobre programas e pacotes usados no laboratório.

Para além disso, a apostila também deve servir como guia introdutório para a aprendizagem de ROS.

É com muito esmero e dedicação que realizamos esta atividade.

2020,
Arthur H. D. Nunes,
Adriano M. C. Rezende
Orientador: Luciano C. A. Pimenta

LABORATÓRIO DE SISTEMAS DE COMPUTAÇÃO E ROBÓTICA - UNIVERSIDADE FEDERAL DE MINAS GERAIS

[HTTP://CORO.CPDEE.UFGM.BR/](http://CORO.CPDEE.UFGM.BR/)



Sumário

I

Noções Básicas

1	Linux	9
1.1	O Sistema Operacional	9
1.2	Instalação	10
1.2.1	Máquina Virtual	10
1.2.2	Dual Boot	10
1.3	Terminal	11
1.3.1	Atalhos	11
1.3.2	Navegação Em Pastas	12
1.3.3	Opções Com Arquivos	12
1.3.4	Permissões	13
1.3.5	Programas em Segundo Plano	14
1.3.6	Instalação	14
2	Programação	17
2.1	Python	17
2.1.1	Visão Geral	17
2.1.2	Ambientes Virtuais	18
2.1.3	IDEs	18
2.1.4	Estrutura de Dados	18
2.1.5	Comandos e Funções	19
2.1.6	Operadores	19
2.1.7	Controle de Fluxo	21

2.2	C++	21
2.2.1	Visão Geral	21
2.2.2	Compilação	21
2.2.3	IDEs	21
2.2.4	Estrutura de Dados	22
2.2.5	Comandos e Funções	22
2.2.6	Operadores	22
2.2.7	Controle de Fluxo	24

II

Robot Operating System

3	Preparação	27
3.1	O Software	27
3.2	Instalação	28
3.3	Área de Trabalho e Pacotes	29
3.4	Navegação	31
4	Utilização	33
4.1	Nós e Tópicos	33
4.2	Criando nós	39
4.2.1	Publicar	39
4.2.2	Subscrever	43
4.2.3	Compilação e Execução	46
4.2.4	Mais Nós - Controlando a turtlesim	48
4.3	Serviços e Parâmetros	56
4.4	Launch	56
4.4.1	Remap e Namespace	56
4.4.2	Criando Arquivos .launch	57
4.5	Múltiplas Máquinas	58
5	Simulação	59
5.1	Stage	59
5.2	RViz	66
5.3	Gazebo	72
5.4	Dispositivos	72
5.4.1	Joystick	72
5.4.2	Câmeras	72
5.4.3	Turtlebot	72
5.5	Simulação final	72

III

Adicionais

Bibliografia	75
Livros	75

Links	75
Repositórios	77
Apêndice I - CORO	79
Instalação	79
Joystick	80
iRobot Create	81
Câmeras	82



Noções Básicas

1	Linux	9
1.1	O Sistema Operacional	
1.2	Instalação	
1.3	Terminal	
2	Programação	17
2.1	Python	
2.2	C++	



1. Linux

O foco desta apostila é a utilização do ROS, portanto este capítulo trará somente o básico do Linux.

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** será utilizada, como mostra a Fig. 1.1.1, porque é, atualmente, a que

possui melhor compatibilidade com o software **ROS Kinetic Kame** Fig. 3.1.2.

1.2 Instalação

Esta seção abordará dois métodos de instalação do Ubuntu 16, caso o usuário ainda não a tenha.

1.2.1 Máquina Virtual

Uma maneira de utilizar o Ubuntu possuindo outros sistemas operacionais é a criação de uma máquina virtual. Desta 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.

Tutorial:

1.2.2 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:

- 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", 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 partitionar 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 Seleciona a língua que deseja e em seguida, em Instalar Ubuntu.
 - 7 Seleccione 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.

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\TimeZoneInformation"
 - 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.

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 com o Gnome Terminal instalado. Outros terminais também podem ser instalados, de acordo com a preferência do usuário. Terminator é uma boa opção.

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 compreendimento 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, se indicarmos a ação:

mkdir <nome>

deve ser feito 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. Deve ficar atento pois Ctrl + C não copia, e sim interrompe a execução no momento, para copiar e colar deve pressionar Ctrl + Shift + C e Ctrl + Shift + V, respectivamente. Os atalhos Ctrl + Shift + N e Ctrl + Shift + T abrem uma nova janela e nova aba, respectivamente. É possível realizar estas ações clicando em "Arquivo"no canto superior direito da janela. Também neste canto, em "Editar"depois "Preferências"é possível configurar cores, fontes e etc. da forma que o usuário mais gostar. Ctrl + "+"aumenta a letra, enquanto que Ctrl + "-"a diminui, também podendo ser feito clicando em "Ver". O atalho tab completa automaticamente o comando ou sugere as opções de completar quando existe mais de uma opções.

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.

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 está 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 traço 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 usado 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 deste 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, esta ação pode ser interpretada como 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 destes arquivos podem ser alteradas. O usuário *root* é o usuário 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 de que pertence. Por padrão, quando não se cria grupos, cada usuário terá seu grupo com seu respectivo nome.

Já os dez primeiros caracteres dizem sobre 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>*. Na qual 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 concedermos 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 as alterações de permissões afete 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
```

1.3.6 Instalação

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

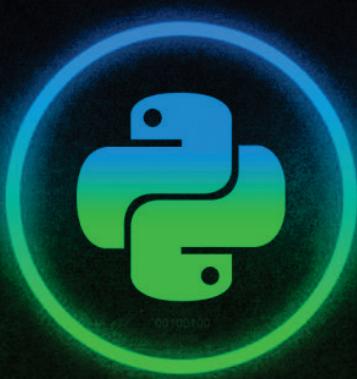
```
$ sudo apt-get <update/install/remove> <programa>
```

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

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



```
print "Hello World"
```

2. Programação

O foco desta apostila é a utilização do ROS, portanto este capítulo trará somente o básico para seu uso. Não aprofundaremos em programação e recomendamos uma boa noção prévia.

2.1 Python

2.1.1 Visão Geral

A linguagem Python foi desenvolvida em 1991 por Guido Van Rossum. Possui várias aplicações, pois é uma linguagem de alto nível, orientada a objetos e possui uma sintaxe elegante e tipagem dinâmica. Pode ser facilmente transformada para uma aplicação por meio da importação de bibliotecas.

É uma linguagem interpretada, assim, não necessita de compilação. Nesta linguagem, a identação é fundamental, o código não irá funcionar se não estiver devidamente identado, pois é este artifício que indica quais estruturas estão subordinadas, diferente de C++ que utiliza chaves para isso.

O Linux, assim como MacOS, a possui, e pode ser acessada utilizando o comando *python*, e para sair *exit()*. Há diferenças entre as versões de Python, por isso, recomendamos a 2.7, que é a versão que o ROS Kinetic Kame utiliza, bem como a versão do Ubuntu 16.04 LTS.

Trocar a versão do Python do seu sistema operacional irá comprometer algumas funções, como abrir o terminal ou instalar programas pelo Ubuntu Software. É um erro comum trocar a versão do python 3 do 3.5 para o 3.6, principalmente quando se está aprendendo. Para resolver este erro pressione Ctrl + Alt + F7 para sair do modo GUI e entrar no modo CLI de seu sistema. Feito isso, selecione a versão recomendada (3.5) após digitar o comando

```
$ sudo update-alternatives -config python3
```

Para prevenir erros como este, é recomendável o uso de ambientes virtuais (VENVs) para a prática da linguagem.

Recomendamos também o uso da IDE PyCharm da JetBrains, que também possui um *venv*. Disponível em: <https://www.jetbrains.com/pycharm/>

2.1.2 Ambientes Virtuais

Para o uso do ROS, não será necessário a criação de ambientes virtuais. São apenas boas práticas para o uso e aprendizado da linguagem.

Além de prevenir erros no python do seu sistema, o ambiente virtual pode nos ajudar a não encher nosso sistema de bibliotecas desnecessárias, bem como prevenir conflitos entre elas e suas versões durante o processo de aprendizagem ou uso da linguagem.

O funcionamento de um ambiente virtual é simples, ele cria cópias de todos os diretórios necessários que o programa Python precisa, assim, as modificações serão feitas neste ambiente, e não globalmente. Para utilizarmos, é preciso instalar o pip e o virtualenv.

```
$ sudo apt install python-pip
$ sudo pip install virtualenv
```

Para criarmos um *virtualenv* e ativá-lo devemos usar respectivamente os comandos:

```
$ virtualenv <nome>
$ source <nome>/bin/activate
```

Note que o nome aparecerá como prefixo sempre que você estiver dentro dele. Podemos também desativá-lo ou removê-lo com os comandos:

```
$ deactivate
$ rm -r <nome>
```

2.1.3 IDEs

IDEs são ambientes de desenvolvimento integrado, ou *integrated development environments*. Esses ambientes possuem ferramentas que facilitam o aprendizado e uso da linguagem. Para o Python, a IDE PyCharm, do desenvolvedor JetBrains é uma boa opção. Possui atalhos para completar o texto, plugins, seu próprio ambiente virtual para execução e interpretação e executa os códigos.

No entanto, durante o uso do ROS, arquivos de diferentes extensões deverão ser criados ou modificados, e para isso, um editor de texto com maior abrangência é recomendada, como o Atom ou Sublime Text.

2.1.4 Estrutura de Dados

As variáveis em Python, por padrão, são objetos. O que explica, por exemplo, o motivo do operador "+" somar inteiros e concatenar strings. Por serem objetivos, facilitam alguns procedimentos usando os seus métodos. Por exemplo, veja esses métodos de strings

```
1 Frase = "Veja a frase"
2 [a, b, c] = Frase.split()
3 print Frase.upper()
4 print a, c
```

Será impresso:

VEJA A FRASE
Veja frase

2.1.5 Comandos e Funções

O comando *import* é utilizado para importar bibliotecas, e será fundamental para a utilização do ROS. É possível importar toda a biblioteca com *import <bib>* ou importar apenas os itens desejados, utilizando *from <bib> import <dados>*. Do primeio modo, se quisermos utilizar algo, deveremos utilizar nome da biblioteca, já do segundo, isto não é necessário. Veja o exemplo do cálculo do cosseno.

```
1 import math
2 a = math.pi
3 x = math.cos(a)
4 print x

1 from math import pi, cos
2 a = pi
3 x = cos(a)
4 print x
```

Nos dois casos, será impresso -1. Para o uso do ROS, devemos importar a biblioteca rospy, bem como os tipos de dados que precisarmos. Será comum em nossos programas cabeçalhos como este:

```
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
```

A primeira linha serve para indicar ao terminal que o código deve ser executado em Python, devemos usá-la em todo programa que fizermos para o ROS.

2.1.6 Operadores

Para alguém que sabe programar, mas talvez não conheça a Python, é prático uma tabela com os operadores da linguagem, para se adaptar a sintaxe. Iremos listar os operadores aritméticos, relacionais, lógicos, atribuição, bit a bit, filiação e identidade.

Tabela 2.1.1: Operadores

Tipo	Operador	Descrição
Aritmético	+	Adição
Aritmético	-	Subtração
Aritmético	*	Multiplicação
Aritmético	/	Divisão
Aritmético	//	Divisão inteira
Aritmético	%	Resto da divisão
Aritmético	**	Potenciação
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor ou igual a
Lógico	not	Não
Lógico	and	E
Lógico	or	Ou
Atribuição	=	Atribuição
Atribuição	+=	Atribuição adição
Atribuição	-=	Atribuição subtração
Atribuição	*=	Atribuição multiplicação
Atribuição	/=	Atribuição divisão
Atribuição	//=	Atribuição divisão inteira
Atribuição	%=	Atribuição resto
Atribuição	**=	Atribuição potenciação
Bit a bit	&	E
Bit a bit		Ou
Bit a bit	^	Ou exclusivo
Bit a bit	~	Complemento de um
Bit a bit	«	Deslocamento à esquerda
Bit a bit	»	Deslocamento à direita
Filiação	in	Está em
Identidade	is	Apontam para o mesmo objetivo

Precedência:

- ()
Parênteses
- **
Potenciação
- ~ + -
Complemento, positivo e negativo (unários)
- * / % //
Multiplicação e divisão
- + -
Adição e subtração
- >> <<
Deslocamento binário
- &
E binário
- ^ |
Não binário e ou binários
- <= < > >=
Maior e menor que
- <> == !=
Igual a e diferente de
- = %= /= //= -= += *= **=
Atribuição
- is
Identidade
- in
Filiação
- not or and
Relacionais

2.1.7 Controle de Fluxo

```
1 xf, yf = raw_input('Digite a coordenada (x,y) desejada: ')
      split()
2 xf, yf = [float(xf) for xf in [xf, yf]]
```

2.2 C++

2.2.1 Visão Geral

C++ é uma extensão da linguagem C. É popular e é a principal língua mãe dos programadores brasileiros. Nela, os códigos são compilados, como Fortran, e por isso, a execução é mais rápida do que em linguagens interpretadas, como Python e MATLAB.

2.2.2 Compilação

2.2.3 IDEs

Geralmente, os ambientes de desenvolvimento integrado (IDEs) de C++ possuem ferramentas de compilação e depuração que ajudam no processo de aprendizagem, construção e revisão de códigos. Duas muito comuns são Dev-C++ e Code Blocks.

No entanto, durante o uso do ROS, arquivos de diferentes extensões deverão ser criados ou

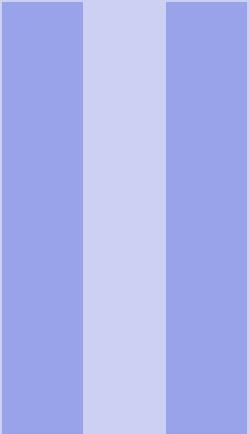
modificados, e para isso, um editor de texto com maior abrangência é recomendada, como o Atom ou Sublime Text.

2.2.4 Estrutura de Dados**2.2.5 Comandos e Funções****2.2.6 Operadores**

Tabela 2.2.1: Operadores

Tipo	Operador	Descrição
Aritmético	+	Adição
Aritmético	-	Subtração
Aritmético	*	Multiplicação
Aritmético	/	Divisão
Aritmético	%	Resto da divisão
Aritmético	++	Pós ou pré incremento
Aritmético	--	Pós ou pré decremento
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	>	Maior que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor ou igual a
Lógico	!	Não
Lógico	&&	E
Lógico		Ou
Atribuição	=	Atribuição
Atribuição	+=	Atribuição adição
Atribuição	-=	Atribuição subtração
Atribuição	*=	Atribuição multiplicação
Atribuição	/=	Atribuição divisão
Atribuição	%=	Atribuição resto
Bit a bit	&	E
Bit a bit		Ou
Bit a bit	^	Ou exclusivo
Bit a bit	~	Complemento de um
Bit a bit	<<	Deslocamento à esquerda
Bit a bit	>>	Deslocamento à direita
Endereçamento	*	Ponteiro
Endereçamento	&	Endereço de
Endereçamento	.	Referência de estrutura
Endereçamento	->	Deferência de estrutura
Escopo	::	Resolução de escopo

2.2.7 Controle de Fluxo



Robot Operating System

3	Preparação	27
3.1	O Software	
3.2	Instalação	
3.3	Área de Trabalho e Pacotes	
3.4	Navegação	
4	Utilização	33
4.1	Nós e Tópicos	
4.2	Criando nós	
4.3	Serviços e Parâmetros	
4.4	Launch	
4.5	Múltiplas Máquinas	
5	Simulação	59
5.1	Stage	
5.2	RViz	
5.3	Gazebo	
5.4	Dispositivos	
5.5	Simulação final	

3. Preparação

3.1 O Software

O ROS é um *framework* para desenvolvimento de softwares robóticos, visa facilitar a criação de complexos e robustos comportamentos de máquinas, provido de bibliotecas, ferramentas, convenções, atalhos, simuladores e tutoriais. O ROS permite a execução de vários programas e estabelece comunicações entre eles. O núcleo do ROS, 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. 3.1.1. A estrutura final é a de um grafo. Omitiremos a representação do *roscore* para simplificação. O programa *talker* indica ao *roscore* que irá enviar um tipo de mensagem, enquanto que o programa *listener* indica que irá receber este mesmo tipo, assim, o *roscore* estabelece uma comunicação entre eles.

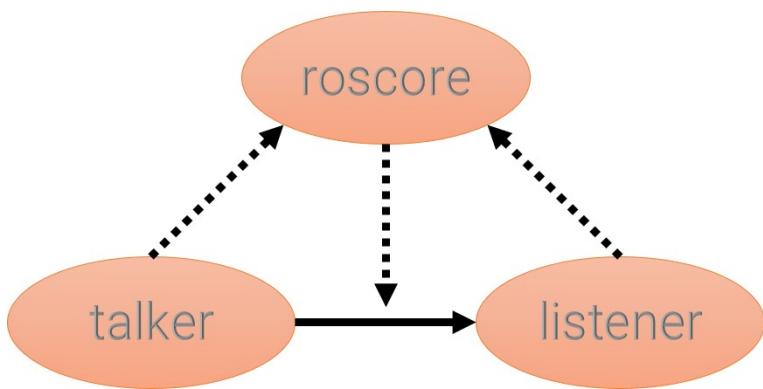


Figura 3.1.1: Grafo Esquemático 1

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. 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.

Aos programas damos os nomes de **nós**, e às comunicações de **tópicos**.

Utilizaremos a versão Kinetic Kame do ROS, Fig. 3.1.2, melhor compatível com nosso sistema operacional Ubuntu 16.04 LTS.



Figura 3.1.2: ROS Kinetic Kame

3.2 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 serão explicados posteriormente.

```
$ 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-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

$ 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
```

Antes da instalação, propriamente dita, deve fazer com que o computador permita a instalação do software de packages.ros.org, o que é o primeiro comando. O comando configura suas chaves de instalação.

Em sequência, deve-se verificar se os pacotes estão atualizados, pelo terceiro comando. O quarto pode demorar um pouco e é o recomendado, instalar o *ros-desktop-full*, mas pode ser substituído, caso o usuário queira, por instalar apenas ROS, *rqt*, *rviz*, e 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, que 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.

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

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

3.3 Área de Trabalho e Pacotes

Para começar a usar o ROS é necessário criar áreas de trabalho ou **workspaces** e pacotes ou **packages**. O tipo recomendado de workspace é o tipo **catkin_make**.

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
```

Sempre que se for utilizar a workspace, como seus nós, deve-se usar o *source* sempre que abrir um novo terminal.

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

O comando pode ser esquecido de ser usado nas várias vezes que deve ser usado. Por isso, caso tenha uma única workspace, ela pode ser sourceda colocando o código no arquivo `.bashrc`, assim como foi feito anteriormente, pelo comando a seguir, ou adicionado manualmente ao arquivo.

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

Os pacotes são pastas nas quais se encontram os arquivos executados pelo ROS. Os pacotes ficam dentro da pasta `src` da workspace e devem conter pelo menos dois arquivos: `CMakeLists.txt` e `package.xml`. A estrutura de pacotes de uma workspace é ilustrada pela Fig. 3.3.1

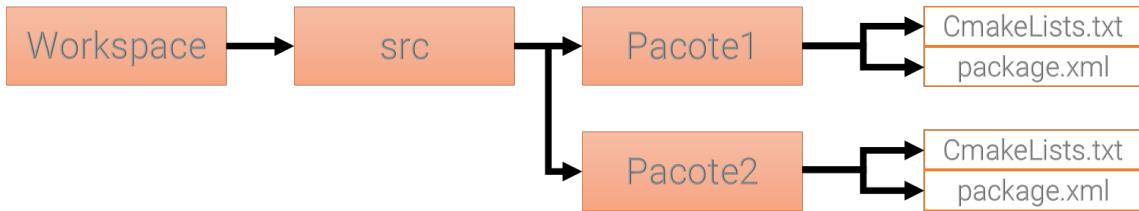


Figura 3.3.1: Estrutura Básica de Pacotes

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

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

O comando de criação de pacote é sucedido do nome do pacote e de suas dependências. O pacote padrão criado para iniciar os tutoriais é

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

Criando assim o pacote `beginner_tutorials`, que depende de `std_msgs`, `rospy` e `roscpp`. Uma dependência pode possuir outras dependências indiretas, para visualizar as dependências recursivamente do pacote pode ser usado:

```
$ rospack depends beginner_tutorials
cpp_common
rostime
roscpp_traits
roscpp_serialization
catkin
genmsg
genpy
message_runtime
gencpp
geneus
gennodejs
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
```

```
ros_environment
rospack
roslib
rospy
```

O arquivo *package.xml* contém as informações meta do pacote, sendo recomendada a alteração apenas da descrição (linha 5) para que outros usuários possam entender de que se trata o pacote. O exemplo ficará:

package.xml

```
1 <?xml version="1.0"?>
2 <package format="2">
3 <name>beginner_tutorials </name>
4 <version>0.1.0</version>
5 <description>The beginner_tutorials package</description>
6
7 <maintainer email="you@yourdomain.tld">Your Name</
     maintainer>
8 <license>BSD</license>
9 <url type="website">http://wiki.ros.org/beginner_tutorials
     </url>
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.

3.4 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. Iremos usar 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 sua 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
```



4. Utilização

4.1 Nós e Tópicos

Como abordado anteriormente, o esquema de nós e tópicos é a representação do funcionamento do software por meio de um grafo. Cada nó é um programa que pode receber ou enviar dados para outros, e a esses dados enviados, chamados de tópicos. Um mesmo programa pode receber e enviar de vários tópicos, e um mesmo tópico receber ou enviar para vários nós, não há restrição. Ao envio de mensagens nomeamos publicar, e ao 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, omitiremos o *roscore*.

Usaremos o pacote ros-tutorials:

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

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

```
$ roscore
...
... logging to /home/arthur/.ros/log/37f2df16-639b-11e9
-9836-fc017cf4f2b/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
* /rosdistro: 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-fc017cf4f2b
process[rosout-1]: started with pid [2644]
started core service [/rosout]

Em um novo terminal:
$ rosrun turtlesim turtlesim_node
Abrirá uma nova janela similar a Fig. 4.1.1
```

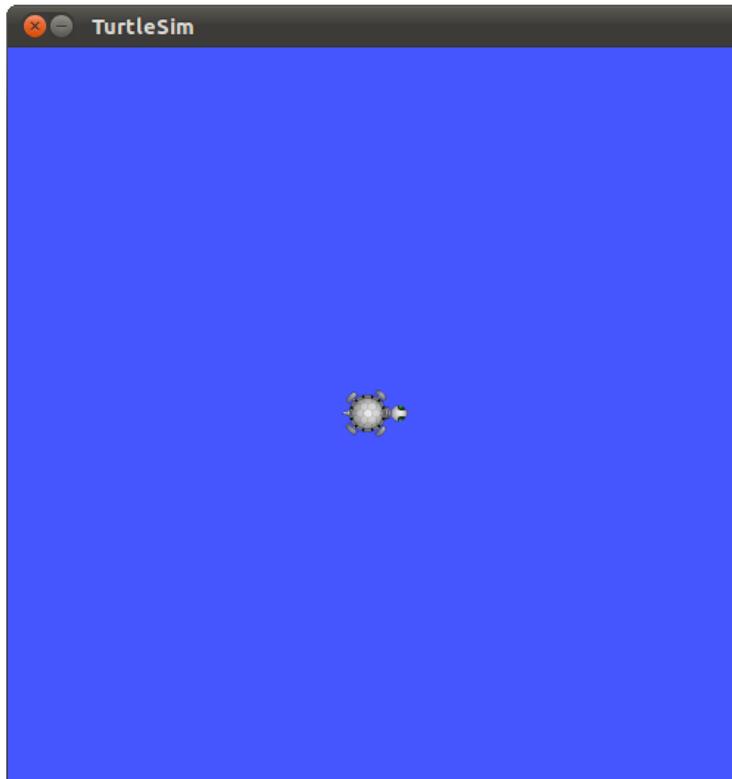


Figura 4.1.1: Primeira Execução de *turtlesim_node*

Foi executado um nó, e é possível verificar pelo comando

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

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, vamos executar outro nó, o *turtle_teleop_key*.

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

[style=Shell] 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 podermos visualizar esse 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
```

Ele pode ser executado com um dos comandos a seguir:

```
$ rqt_graph
$ rosrun rqt_graph rqt_graph
```

Será aberto uma nova janela similar a Fig. 4.1.2:



Figura 4.1.2: Grafo de Execução dos Nós *turtlesim*

Similar ao *rosnode* para os nós, há o *rostopic* para os tópicos, e pode-se ver agora os tópicos existentes com:

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

Outra função útil do *rostopic* - é possível ver todas digitando "rostopic" e pressionando Tab duas vezes ou usando o comando *rostopic -h* - 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.

Continue com, *turtlesim_node* e *turtle_teleop_key* em execução. O quadrado onde a

tartaruga se move tem origem no canto inferior esquerdo e possui aproximadamente 11x11 de tamanho. Podemos utilizar o comando a seguir para verificar a posição em um novo terminal

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

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

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

A medida em que a tartaruga se move, a posição é alterada. Em um novo terminal, podemos dar um echo também no tópico *cmd_vel*, no qual os registros de velocidade são publicados. Note que neste caso, novas mensagens são impressas, apenas quando as setas são pressionadas:

```
$ 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 sozinho, mas executando novamente ou clicando em atualizar é possível observar algo como a Fig. 4.1.3. O comando *rostopic* ainda pode ser executado

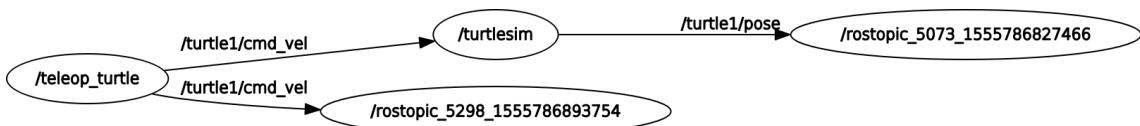


Figura 4.1.3: Grafo de Execução dos Nós turtlesim e rostopic

para publicar diretamente em um tópico. Usaremos isto para dar comandos de velocidades para a tartaruga. Para isso, precisamos saber o tipo de mensagem que devemos publicar:

```
$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
$ rosmsg 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, publicamos a mensagem. A tecla Tab pode ser útil para completar os comandos.

```
$ 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. 4.1.4 Para postar continuamente a mensagem e

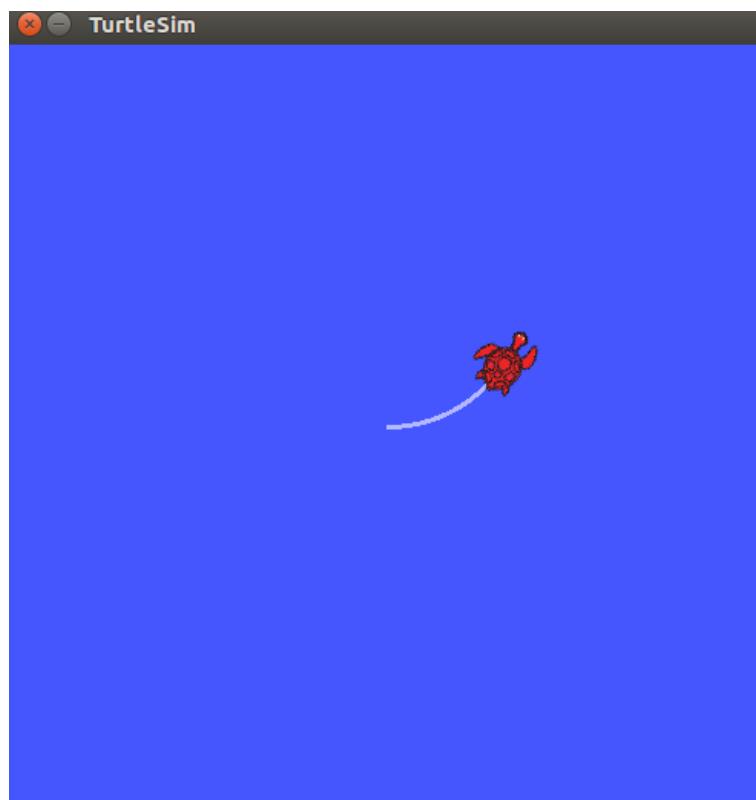


Figura 4.1.4: Publicação direta em cmd_vel

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]',
```

Podemos 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 "+".

Vamos fazer isso com `/turtle1/pose/x` e `/turtle1/pose/y`.

```
$ rosrun rqt_plot rqt_plot
```

Um gráfico dinâmico com a posição será executado, e as medidas definidas digitando-os no espaço em branco clicando em "+". Veja a Fig. 4.1.5.

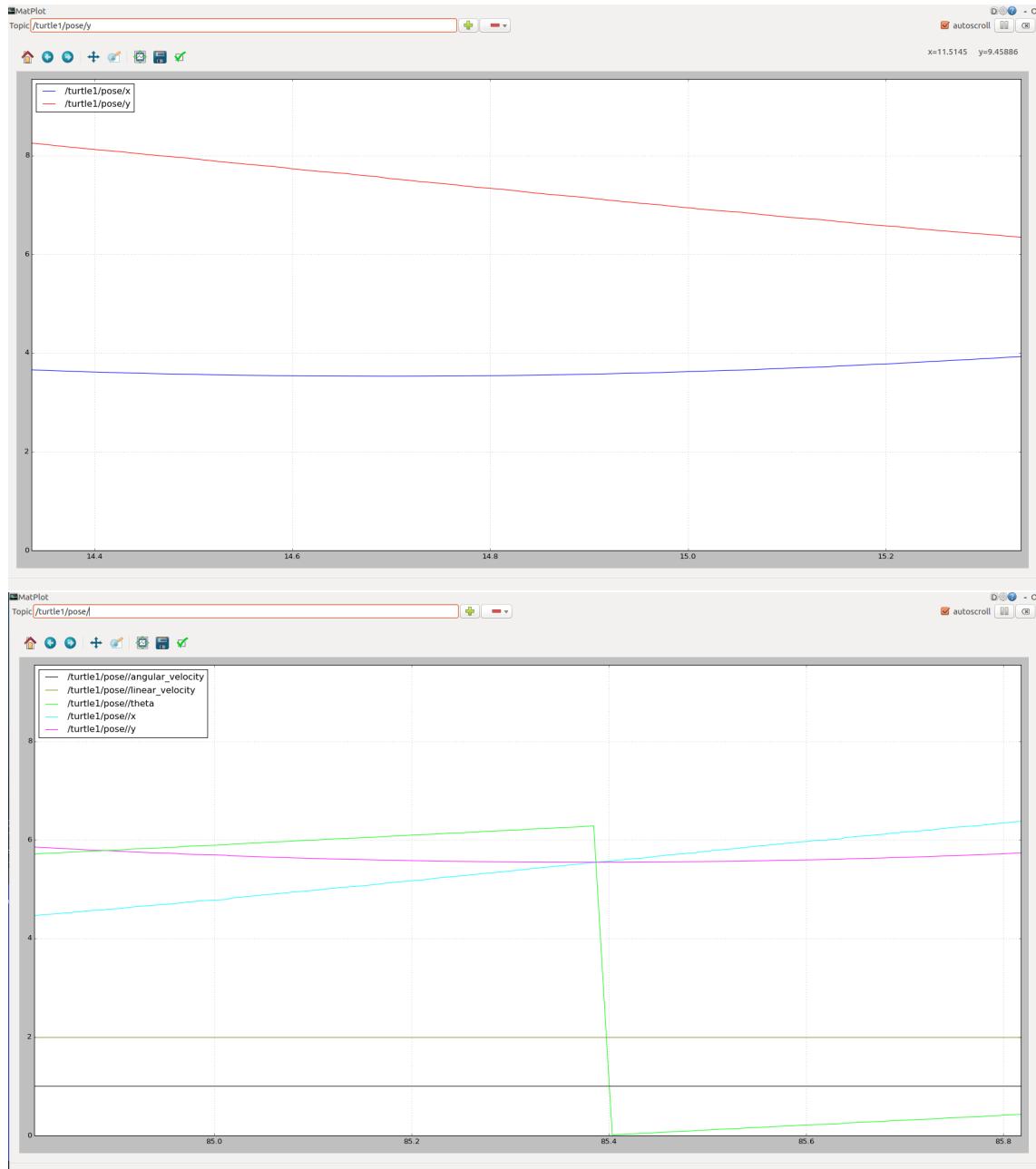


Figura 4.1.5: Uso do `rqt_plot`

4.2 Criando nós

Iremos criar agora os exemplos clássicos: *talker* e *listener*, o equivalente ao primeiro programa *Hello World* no ROS. Utilizaremos 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*. Por isso:

```
$ rosdep init
$ rosdep update
$ roscd beginner_tutorials
$ mkdir src
$ mkdir scripts
```

4.2.1 Publicar

Em Python, lembre-se de guardar o arquivo dentro de *beginner_tutorials/scripts* e em C++ *beginner_tutorials/src*.

A primeira linha será a mesma em todos os nós em Python, serve para garantir que serão interpretados com a linguagem correta. A linha 3 também será obrigatória para todo programa para o ROS, é para importarmos 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()* que executa todo o procedimento de publicação, que será explicado daqui a pouco. A linha 16 é similar a declaração da função *int main* em C++, é o procedimento principal, que chama a função *talker()*, na 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 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) # 10 hz
```

```

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++, serva 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. O primeiro é 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 enfileirando 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, o nó for expulso da rede por outro com o mesmo nome, *ros::shutdown()* foi chamado por outra parte da aplicação ou 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, servirá apenas para se fossemos subscrever em algum outro nó. 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 <iostream>
5
6 /**
7  * This tutorial demonstrates simple sending of messages
8  * over the ROS system.
9 */
10 int main(int argc, char **argv)
11 {

```

```
11  /**
12   * The ros::init() function needs to see argc and argv
13   * so that it can perform
14   * any ROS arguments and name remapping that were
15   * provided at the command line.
16   * For programmatic remappings you can use a different
17   * version of init() which takes
18   * remappings directly, but for most command-line
19   * programs, passing argc and argv is
20   * the easiest way to do it. The third argument to
21   * init() is the name of the node.
22   *
23   * You must call one of the versions of ros::init()
24   * before using any other
25   * part of the ROS system.
26   */
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
38 ros::NodeHandle n;
39
40 /**
41  * The advertise() function is how you tell ROS that
42  * you want to
43  * publish on a given topic name. This invokes a call
44  * to the ROS
45  * master node, which keeps a registry of who is
46  * publishing and who
47  * is subscribing. After this advertise() call is made,
48  * the master
49  * node will notify anyone who is trying to subscribe
50  * to this topic name,
51  * and they will in turn negotiate a peer-to-peer
52  * connection with this
53  * node. advertise() returns a Publisher object which
54  * allows you to
55  * publish messages on that topic through a call to
56  * publish(). Once
57  * all copies of the returned Publisher object are
58  * destroyed, the topic
59  * will be automatically unadvertised.
60  *
```

```
42     * The second parameter to advertise() is the size of
43     * the message queue
44     * used for publishing messages. If messages are
45     * published more quickly
46     * than we can send them, the number here specifies how
47     * many messages to
48     * buffer up before throwing some away.
49     */
50
51     ros::Publisher chatter_pub = n.advertise<std_msgs::
52         String>("chatter", 1000);
53
54     ros::Rate loop_rate(10);
55
56     /**
57     * A count of how many messages we have sent. This is
58     * used to create
59     * a unique string for each message.
60     */
61
62     int count = 0;
63     while (ros::ok())
64     {
65         /**
66         * This is a message object. You stuff it with data,
67         * and then publish it.
68         */
69         std_msgs::String msg;
70
71         std::stringstream ss;
72         ss << "hello world " << count;
73         msg.data = ss.str();
74
75         ROS_INFO("%s", msg.data.c_str());
76
77         /**
78         * The publish() function is how you send messages.
79         * The parameter
80         * is the message object. The type of this object
81         * must agree with the type
82         * given as a template parameter to the advertise<>()
83         * call, as was done
84         * in the constructor above.
85         */
86         chatter_pub.publish(msg);
87
88         ros::spinOnce();
89
90         loop_rate.sleep();
91         ++count;
```

```
81     }
82
83
84     return 0;
85 }
```

4.2.2 Subscrever

Em Python, lembre-se de guardar o arquivo dentro de *begginer_tutorials/script* e em C++ *begginer_tutorials/src*.

Neste caso, temos duas funções declaradas, a que realiza os procedimentos de início de nó *listener()*, similar a *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 iremos usar, neste caso, subscrever.

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 subscrever 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.

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
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
```

```

14     # run simultaneously .
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
18
19     # spin() simply keeps python from exiting until this
20     # node is stopped
21     rospy.spin()
22
23 if __name__ == '__main__':
24     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*. Este método indica que subscreverá no tópico *chatter*, com um tamnho 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 /**
15  * The ros::init() function needs to see argc and argv
16  * so that it can perform
17  * any ROS arguments and name remapping that were
18  * provided at the command line.
19  * For programmatic remappings you can use a different
20  * version of init() which takes

```



```

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 }
```

4.2.3 Compilação e Execução

Criados os códigos, iremos executá-los. Para os nós gerados em Python, é preciso conceder a permissão de execução, que pode ser dada da seguinte forma:

```

$ roscd beginner_tutorials/scripts
$ chmod +x talker.py
$ chmod +x listener.py
$ cd ~/catkin_ws
$ 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 concedida a permissão, ele deve rodar, 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 sempre recompilar com *catkin_make*.

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 podemos executar nossos nós (lembrando que é preciso dar *roscore* e *source* da workspace).

Primeiro iremos executar 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
$ rosrun 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*.

Fechando os nós de Python (Ctrl + C), podemos 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
```

Podemos inclusive, executar um em C++ e outro em Python, ou até mesmo os quatro de uma só vez. Vamos executá-los e abrir a ferramente *rqt_graph* para visualizarmos melhor:

```
$ 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*, podemos 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. 4.2.1.

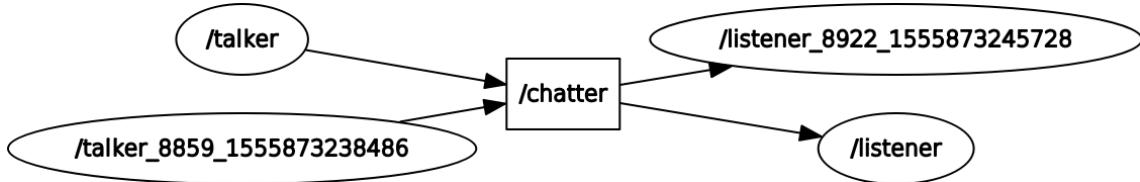


Figura 4.2.1: Grafo de dois talker e dois listener

4.2.4 Mais Nós - Controlando a turtlesim

Para fixar a ideia de construção de nós, iremos agora criar outros nós, em Python, para controlarmos a tartaruga turtlesim.

Tenhamos como primeiro objetivo mandar a tartaruga para um ponto qualquer dentro de seu quadrado. Para isso, iremos precisar publicar dados de velocidade e subscrever dados da posição da tartaruga. Esses dados, encontram-se, respectivamente, nos tópicos */turtle1/cmd_vel* e */turtle1/pose*, podemos verificar executando o *turtlesim_node*, *rostopic list* e *rostopic echo*.

Para o comando de velocidade, precisamos 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:

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

$$\omega = \text{atan}2(\Delta Y, \Delta X)$$

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

O procedimento será realizado até que a tartaruga chegue em uma proximada boa 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
31     while not rospy.is_shutdown():
32         xf, yf = raw_input('Digite a coordenada (x,y) '
33                             'desejada: ').split()
34         xf, yf = [float(xf) for xf in [xf, yf]]
35         if xf > 11 or xf < 0:
36             rospy.loginfo('\033[31mCoordenadas invalidas, '
37                         'digite novamente...\033[m')
38             continue
39         if yf > 11 or yf < 0:
40             rospy.loginfo('\033[31mCoordenadas invalidas, '
41                         'digite novamente...\033[m')
42             continue
43         while abs(x0-xf) > Err or abs(y0-yf) > Err:
44             vel_msg.angular.z = atan2((yf-y0),(xf-x0)) -
45             Theta0
46             pub.publish(vel_msg)
47             rate.sleep()
```

```

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

```

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

A função *atribuirvalorpos* 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 onde a 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 são limitações grotescas da velocidade linear para que a tartaruga não mova muito rápido para frente.

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

Com isso podemos executar nosso programa e observar o resultado:

```

$ cd ~/catkin_ws/src/beginner_tutorials/scripts/
$ chmod +x controle_vel.py
$ cd ~/catkin_ws
$ rosrun turtlesim turtlesim_node
# Novo terminal
# source devel/setup.bash
$ rosrun 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:

```

Podemos usar uma técnica de controle mais robusta.

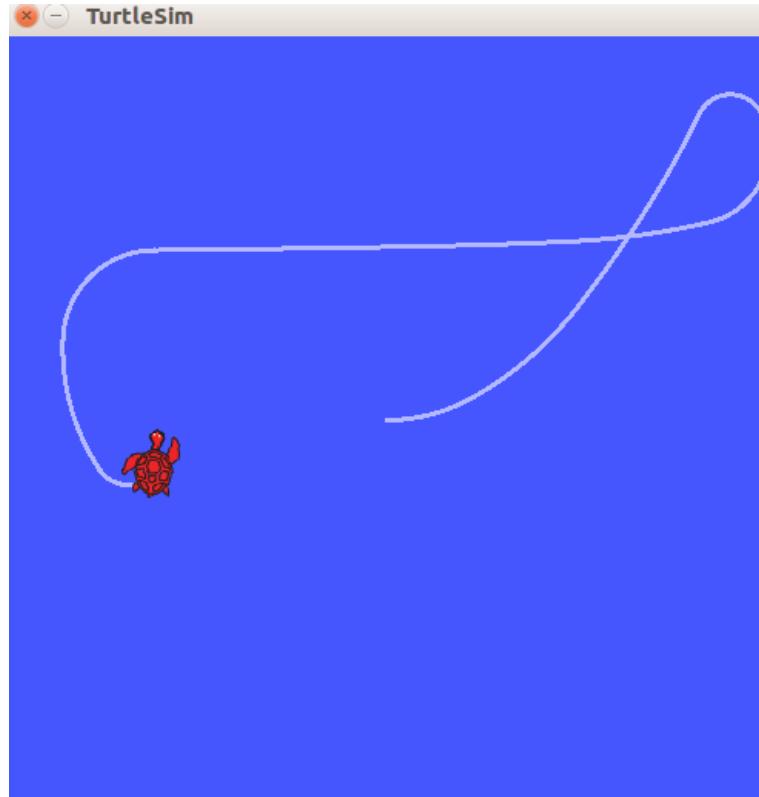


Figura 4.2.2: Movimento da tartaruga

O objetivo é o mesmo, porém utilizando uma técnica melhor, chamada *feedback linearization*. 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:

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

$$\begin{pmatrix} U_x \\ U_y \end{pmatrix} = U = V_{ref} + K \times P$$

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

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.

Implementamos 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)\ndesejada: ').split()
29         xf, yf = [float(xf) for xf in [xf, yf]]
30         if xf > 11 or xf < 0 or yf > 11 or yf < 0:
31             rospy.loginfo('\033[31mCoordenadas invalidas,\n            digite novamente...\033[m')
32             continue
33         lim = 0
34         while abs(xf - x0) > Err or abs(yf - y0) > Err:
35             vel_msg.linear.x = k * (cos(T) * (xf - x0) +
36                                     sin(T) * (yf - y0))
37             vel_msg.angular.z = k * (-(sin(T) * (xf - x0)) /
38                                     d + (cos(T) * (yf - y0)) / d)
39             if vel_msg.linear.x > vel_lim.linear.x:
40                 vel_msg.linear.x = vel_lim.linear.x
41             elif vel_msg.linear.x < -vel_lim.linear.x:
42                 vel_msg.linear.x = -vel_lim.linear.x
43             if vel_msg.angular.z > vel_lim.angular.z:
44                 vel_msg.angular.z = vel_lim.angular.z
45             elif vel_msg.angular.z < -vel_lim.angular.z:
46                 vel_msg.angular.z = -vel_lim.angular.z
47             pub.publish(vel_msg)
48             rate.sleep()
49     if __name__ == '__main__':
50         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 executarmos:

```
$ 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:
```

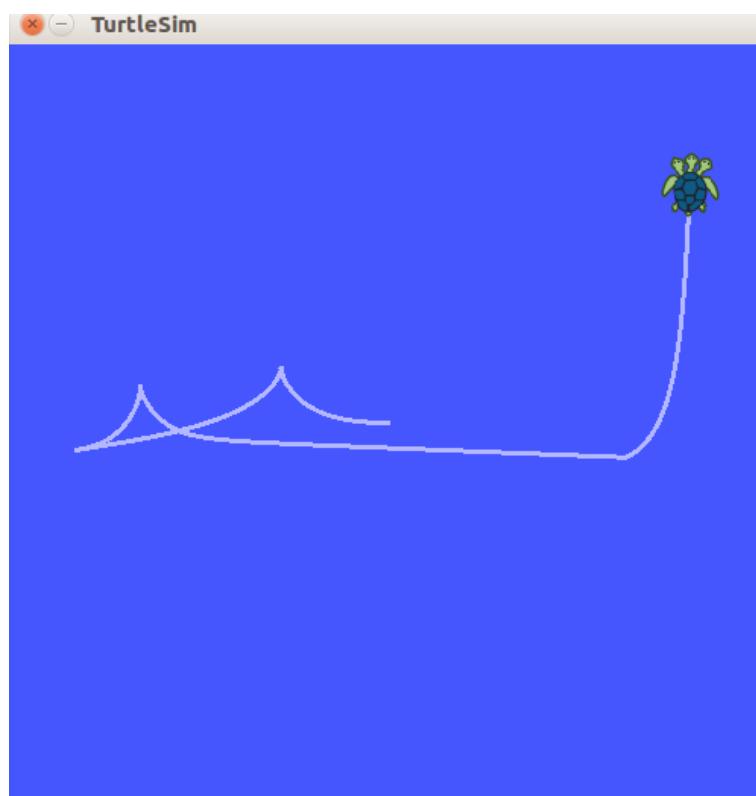


Figura 4.2.3: Movimento da tartaruga

Podemos agora, mudar o ponto alvo com o tempo, fazendo com que a tartaruga persiga uma curva parametrizada no tempo. Desta forma, o Vref será diferente de zero. Para isso, precisamos definir a expressão para o ponto alvo. Iremos utilizar uma elipse paramétrica como exemplo:

$$x_f = a \times \cos(w \times t) + c_x$$

$$y_f = b \times \sin(w \times t) + c_y$$

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

Sendo assim, Vref será a velocidade desse ponto, que é a derivada deles em relação ao tempo:

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

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

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)
20     rospy.Subscriber('/turtle1/pose', Pose, callback)
21     rate = rospy.Rate(20)
22     vel_msg = Twist()
23     cx, cy, rx, ry = raw_input('Digite o centro (x,y) e os
24                                 semieixos (a,b)').split()
25     cx, cy, rx, ry = [float(i) for i in [cx, cy, rx, ry]]
26     t = 0
27     w = 0.4
28     while not rospy.is_shutdown():
29         t += 0.05
30         vel_msg.linear.x = vrefx
31         vel_msg.linear.y = vrefy
32         vel_msg.angular.z = w
33         pub.publish(vel_msg)
34
35 if __name__ == '__main__':
36     iniciar()

```

```
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) *
34                           Vy) / d
35     rate.sleep()
36     pub.publish(vel_msg)
37 if __name__ == '__main__':
38     try:
39         iniciar()
40     except rospy.ROSInterruptException:
41         pass
```

Executamos e observamos o resultado:

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

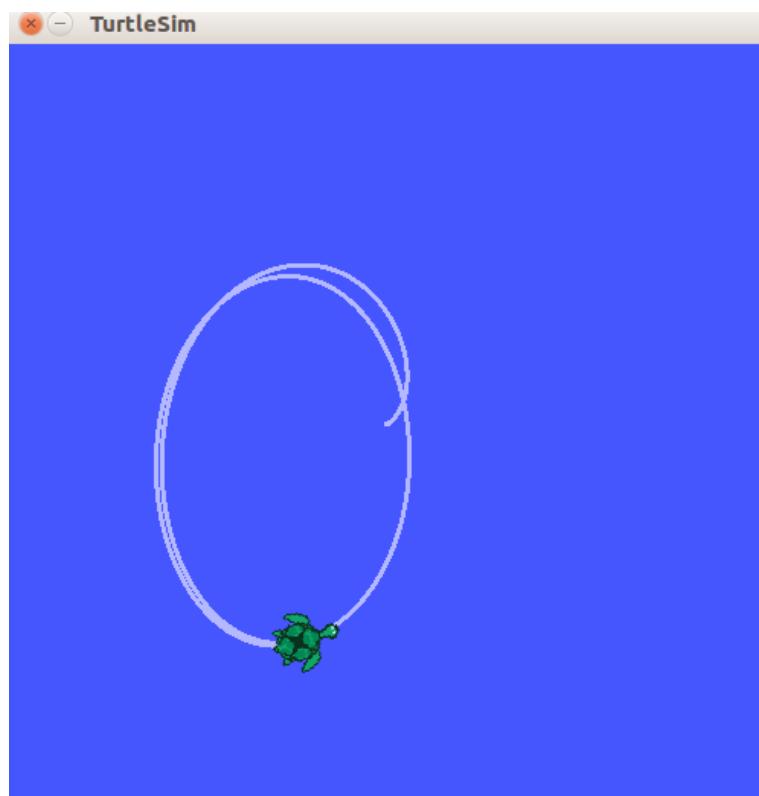


Figura 4.2.4: Movimento da tartaruga

4.3 Serviços e Parâmetros

4.4 Launch

Em sistemas mais complexos, há vários programas parem serem executados, e executá-los 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* e *namespaces*. A este arquivo, chamamos de **launch**. Geralmente, é salvo em uma pasta chamada *launch*, dentro do pacote.

4.4.1 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 é definido o argumento *anonymous=True* nos nós 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 as outras estruturas estarão dentro.

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 39:

```
$ rosrun 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 talker e dois listener como a Fig. 4.2.1 da página 48, porém sem causar interferência entre eles:

```
# Em terminais diferentes
$ rosrun beginner_tutorials talker chatter:=chat1 __name
:=talker1
$ rosrun beginner_tutorials listener chatter:=chat1
__name:=listener1
$ rosrun beginner_tutorials talker chatter:=chat2 __name
:=talker2
$ rosrun beginner_tutorials listener chatter:=chat2
__name:=listener2
```

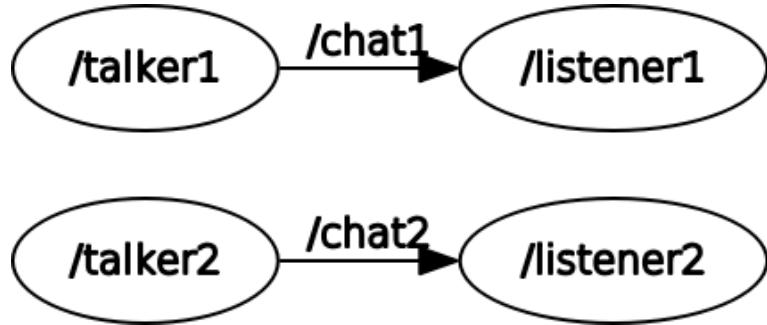


Figura 4.4.1: Grafo dois talker e dois listener sem interferência

4.4.2 Criando Arquivos .launch

Finalmente, pode-se criar o arquivo launch para que execute os nós, faça os remaps e namespaces desejados.

Todo o 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 de uma vez 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="turtle_teleop_key">
7         </node>
8
9 </launch>

```

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

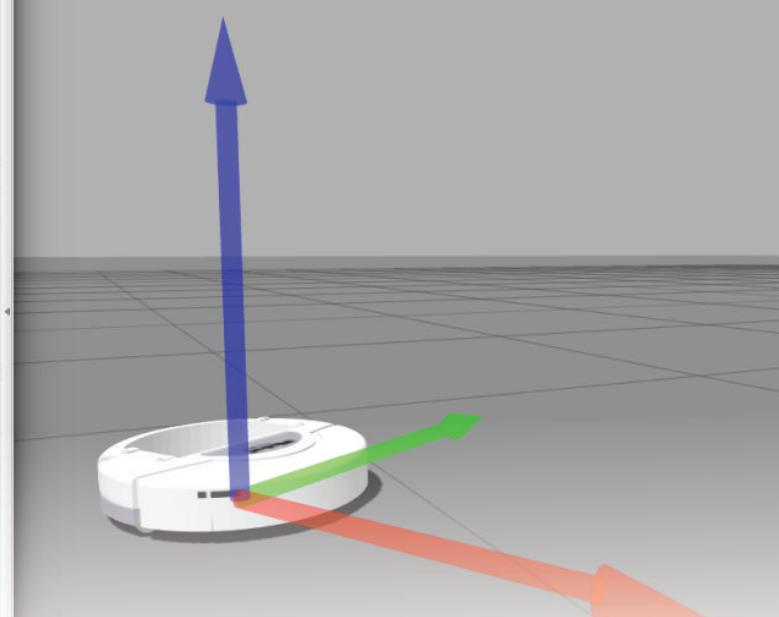
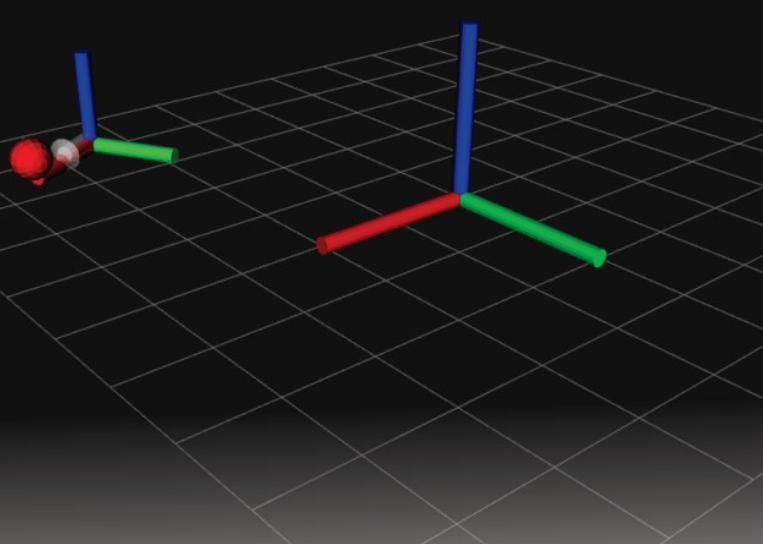
```

1 <launch>
2
3     <group ns="turtlesim1">
4         <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
5     </group>
6
7     <group ns="turtlesim2">
8         <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
9     </group>

```

```
10
11      <node pkg="turtlesim" name="mimic" type="mimic">
12          <remap from="input" to="turtlesim1/turtle1"/>
13          <remap from="output" to="turtlesim2/turtle1"/>
14      </node>
15
16  </launch>
```

4.5 Múltiplas Máquinas



5. Simulação

O ROS possui ótimas ferramentas de simulações. No entanto, é importante conhecer as diferenças das simulações para o real. Nas simulações, por exemplo, a odometria é muito precisa, enquanto que, no real, ela acumula erros e dependendo do método pode ser bastante imprecisa.

Como simuladores, há dois principais, o Stage e o Gazebo. O primeiro deles é mais enxuto e comporta sistemas bidimensionais. Já o segundo, é mais completo e pode demandar um pouco de esforça da CPU do computador. Para além disso, há ainda uma ferramenta de visualização muito potente, o RViz. Com ele é possível acompanhar sua simulação visualizando o mapa, marcadores, eixos, frames, etc.

5.1 Stage

A utilização do simulador Stage será exemplificada. É possível executar o simulador da seguinte forma:

```
$ rosrun 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 utilizado 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 pixels brancos e pretos como a Fig. 5.1.1. Feito isso, o arquivo *world* contém o

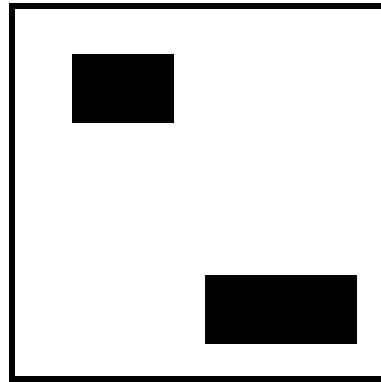


Figura 5.1.1: Arquivo *map_1.png*

seguinte código:

```

map_1.world

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
51 resolution 0.02
52
53 interval_sim 50 # simulation timestep in milliseconds
54
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:

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

Abrirá uma janela como a Fig. 5.1.2. Por fim, basta controlar o robô com um nó como os

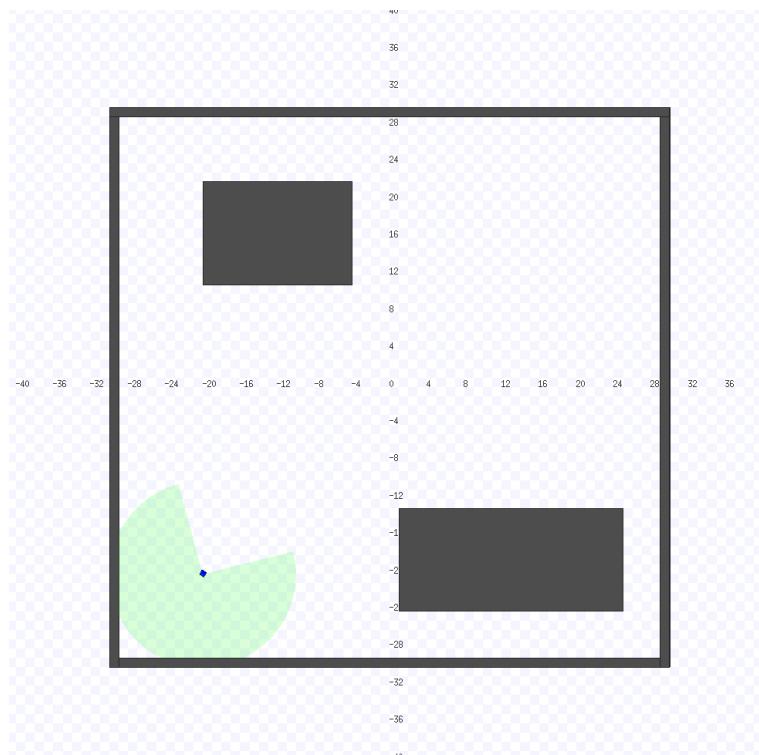


Figura 5.1.2: Execução do Stage

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
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 # Frequencia de simulacao no stage
14 global freq
15 freq = 20.0 # Hz
16 # Velocidade de saturacao
17 global Usat
18 Usat = 5
19 # Estados do robo
20 global x_n, y_n, theta_n
21 x_n = 0.1 # posicao x atual do robo
22 y_n = 0.2 # posicao y atual do robo
23 theta_n = 0.001 # orientacao atual do robo
24 # Estados do robo
25 global x_goal, y_goal
26 x_goal = 0
27 y_goal = 0
28 # Relativo ao feedback linearization
29 global d
30 d = 0.80
31 # Relativo ao controlador (feedforward + ganho proporcional
32     )
33 global Kp
34 Kp = 1
35 # Rotina callback para a obtencao da pose do robo
36 def callback_pose(data):
37     global x_n, y_n, theta_n
38     x_n = data.pose.position.x # posicao 'x' do robo
39         no mundo
40     y_n = data.pose.position.y # posicao 'y' do robo
41         no mundo
42     x_q = data.pose.orientation.x
43     y_q = data.pose.orientation.y
44     z_q = data.pose.orientation.z
45     w_q = data.pose.orientation.w
```

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

```

    rviz
83  pub_rviz_ref = rospy.Publisher("/  

     visualization_marker_ref", Marker, queue_size=1) #  

     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, em um novo terminal, o nó, é observado a convergência do robô para o ponto desejado - no caso a seguir, (0,0) - por feedback linearization, técnica vista na página 51.

```
$ rosrun 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>
```

5.2 RViz

Usando o exemplo anterior como modelo para ser visualizado no RViz, é necessário instalar um pacote para visualização do mapa.

```
$ 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. 5.1.1 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 inciais 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: 0.20000003
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: 1.2000005
Name: Axes
Radius: 0.20000003
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: 000000
  ff00000000fd000000040000000000000016a000002eefc0200000008fb000000

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.

Ao executar os quatro nós, é possível observar a simulação também no RViz, Fig. 5.2.1.

```
$ rosrun stage_ros stageros -d ./src/example/wods/map_1.world
$ rosrun rviz rviz -d ./src/example/rviz/rviz_cfig.rviz
$ rosrun map_server map_server ./src/example/ma/map_1.yaml
$ rosrun example example_node.py 0 0
```

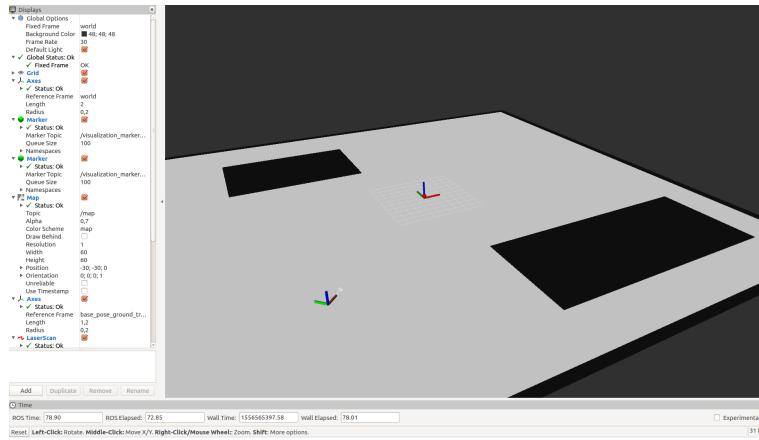


Figura 5.2.1: Visualização da Simulação Stage/RViz

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 controller node
                args="x_goal
                y_goal"-->
14 <node pkg = "example" name = "example_node" type =
    "example_node2.py" args="1 0 2 3" 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>
```

5.3 Gazebo

```

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

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>
```

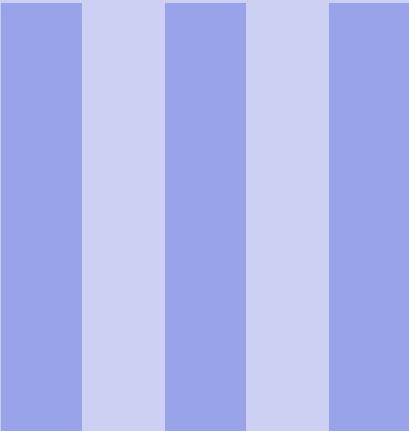
5.4 Dispositivos

5.4.1 Joystick

5.4.2 Câmeras

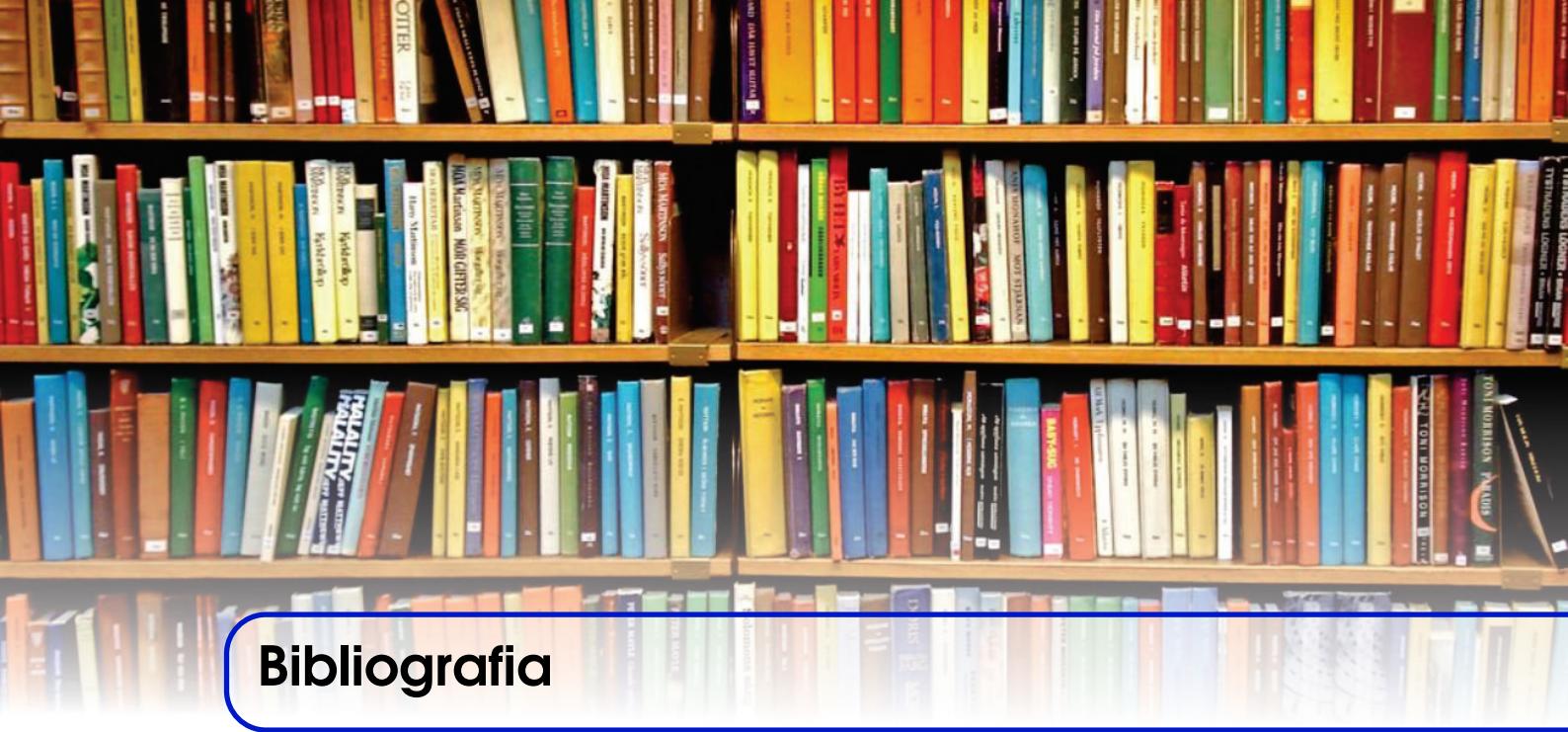
5.4.3 Turtlebot

5.5 Simulação final



Adicionais

Bibliografia	75
Livros	
Links	
Repositórios	77
Apêndice I - CORO	79
Instalação	
Joystick	
iRobot Create	
Câmeras	



Bibliografia

Livros

- Morgan Quigley, Brian Gerkey & William D. Smart (2015) "Programming Robots with ROS" Primeira Edição.
- R. Patrick Goebel (2014) "ROS BY EXAMPLE" Version 1.04 for ROS Hydro

Links

- <https://www.tutorialspoint.com/>
- <http://wiki.ros.org/ROS/Tutorials>
- <http://www.mundobuntu.com.br/>
- <https://www.vivaolinux.com.br/>
- <https://stackoverflow.com/>
- <https://github.com/>



Repositórios

```
https://github.com/joelillo/create_simulator/tree/master/model  
// modelo irobot  
https://bitbucket.org/osrf/gazebo_tutorials/src // geral  
https://github.com/turtlebot/turtlebot_create // turtlebot para kine
```




Apêndice I - 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 10.

- 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", 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 Seleciona 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 28.

```
$ 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-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

$ 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 encontrados 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. Plague-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 e quatro computadores netbooks brancos no laboratório.

Para conectar-se à um robô deve: Verifique se os robôs estão carregados e, se possível, carregue-os depois de usá-los.

- 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 *rosrun 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 se 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 *rosrun 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, estão impressos e pregados em cada um dos branquinhos.

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