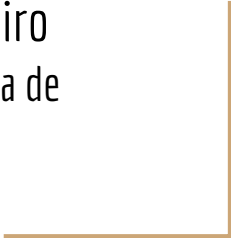


Introdução ao ROS e ao simulador Gazebo

Luciano do Vale Ribeiro
baseado nas notas de aula de
Rafael G. Braga



Bibliografia

- Livro “A gentle introduction to ROS”, que pode ser baixado gratuitamente no link: <https://cse.sc.edu/~jokane/agitr/>
- Documentação oficial do ROS: <https://wiki.ros.org>
- Documentação oficial do Gazebo: <http://gazebo-sim.org/>
- Tutorial “Robotic Simulation with ROS and Gazebo”:
<http://www.generationrobots.com/blog/en/2015/02/robotic-simulation-scenarios-with-gazebo-and-ros/>

Aula 1: Introdução ao ROS

ROS

ROS - Robot Operating System

ROS é um framework (conjunto de programas e ferramentas) de código aberto desenvolvido para servir como base em aplicações de robótica.

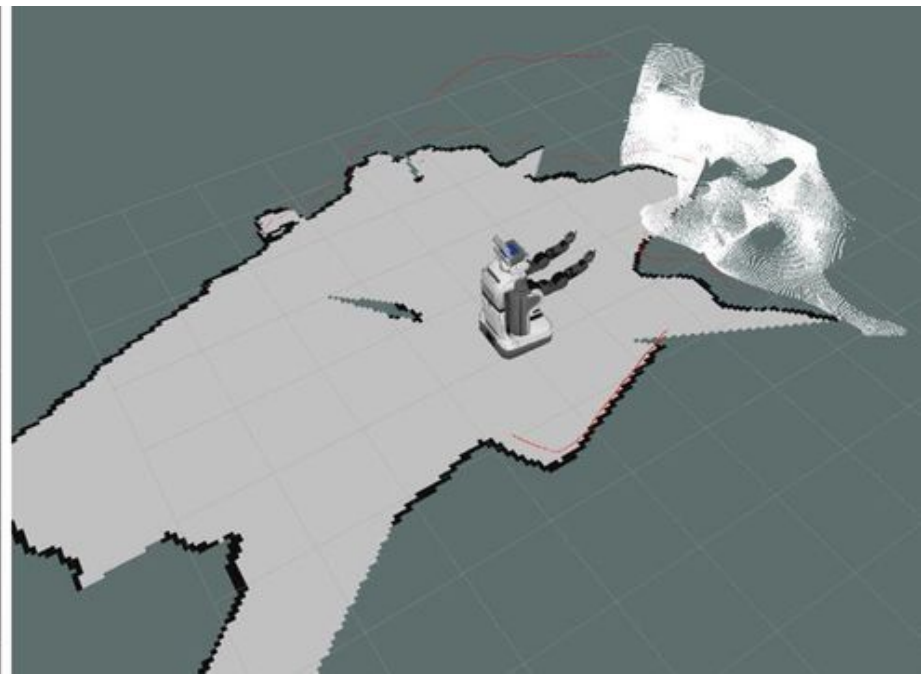
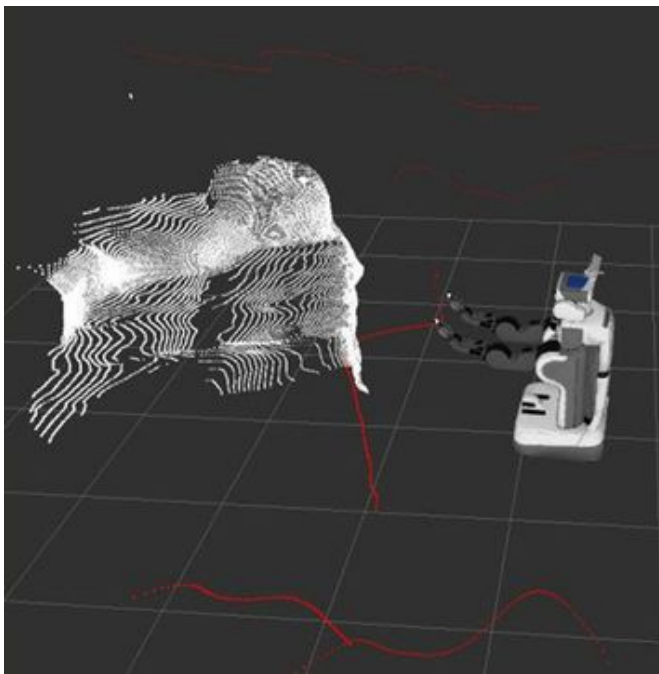
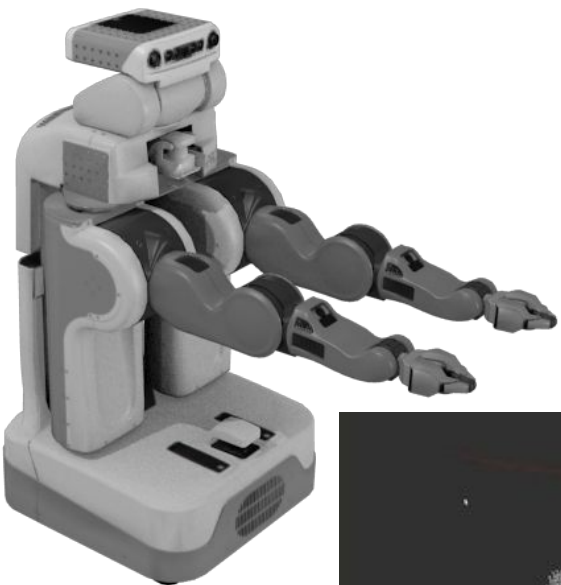
Possui características de um sistema operacional fornecendo diversos serviços como abstração de hardware, implementação de funções comumente utilizadas, passagem de mensagens entre processos, gerenciamento de pacotes, entre outros.

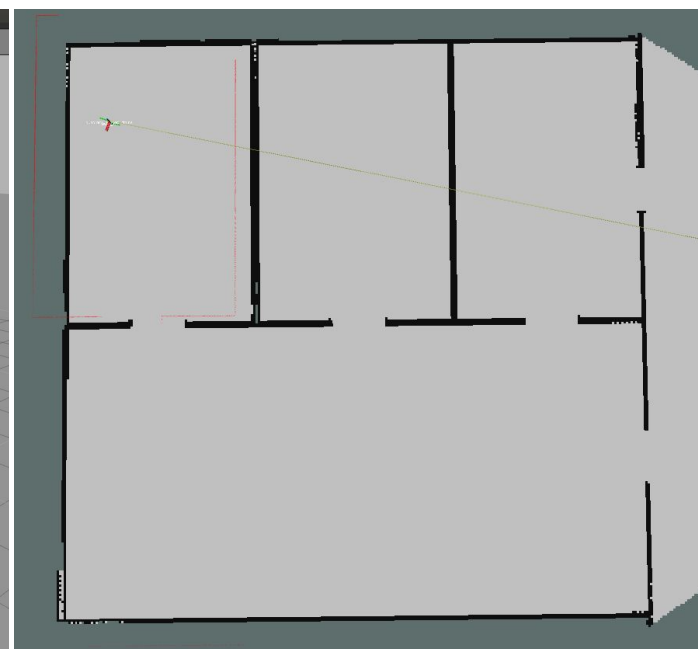
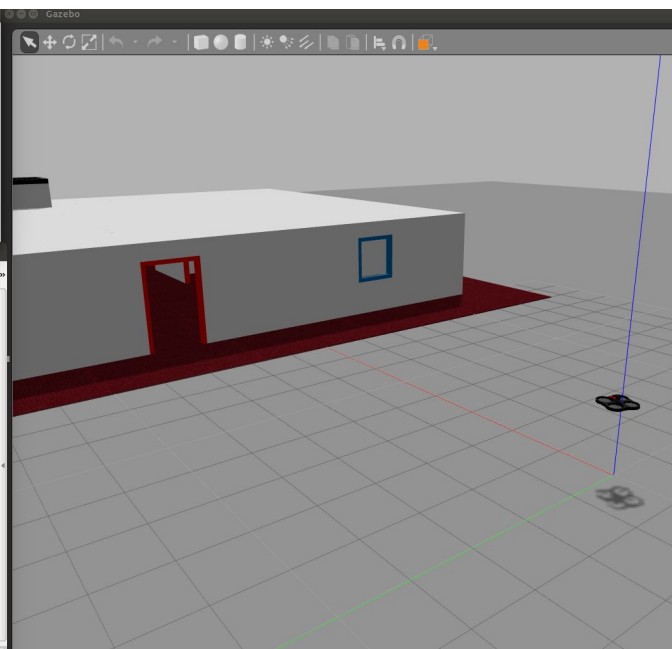
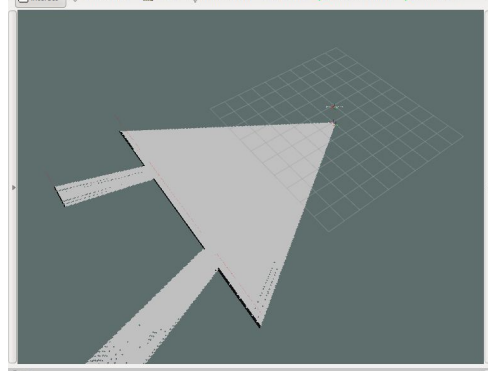
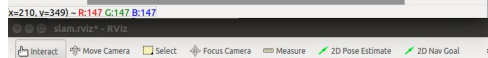
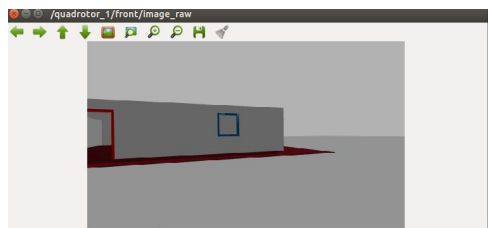
Também fornece bibliotecas e ferramentas para criar código que seja capaz de ser executado através de várias máquinas simultaneamente.

Vantagens do ROS

- Computação distribuída: O ROS permite a criação com facilidade aplicações que são executadas em várias máquinas simultaneamente.
- Reutilização de software: Muitas estruturas e algoritmos padrão estão disponíveis no ROS.
- Teste rápido: O ROS tem ferramentas que facilitam e agilizam o processo de teste do software desenvolvido.
- Grande comunidade de desenvolvedores ativa.

Pode ser programado em C++, Python, Java, entre outras.







Componentes Principais



Infraestruturas de Comunicação

- Transporte de mensagens entre processos (nós)
 - Com uso de tópicos, processos se comunicam através de um canal assíncrono no estilo *publisher/subscriber*.
- Gravar e reproduzir mensagens
 - Mensagens transmitidas por um nó A e recebidas por um nó B podem ser gravadas, e posteriormente reproduzidas. Facilita *debug*.
- Chamada de procedimento remoto (*RPC, Remote Procedure Call*)
- Servidor de Parâmetros centralizado.
 - Permite armazenagem de valores (inteiros, strings) identificados por um nome.

Intalação do ROS

- As instruções encontram-se no site:
<http://wiki.ros.org/kinetic/Installation>
- Instalar a versão **ros-kinetic-desktop-full**
- No nosso curso utilizaremos alguns pacotes adicionais:

```
sudo apt-get update
```

```
sudo apt-get install ros-kinetic-ros-control
```

```
sudo apt-get install ros-kinetic-gazebo-ros-pkgs
```

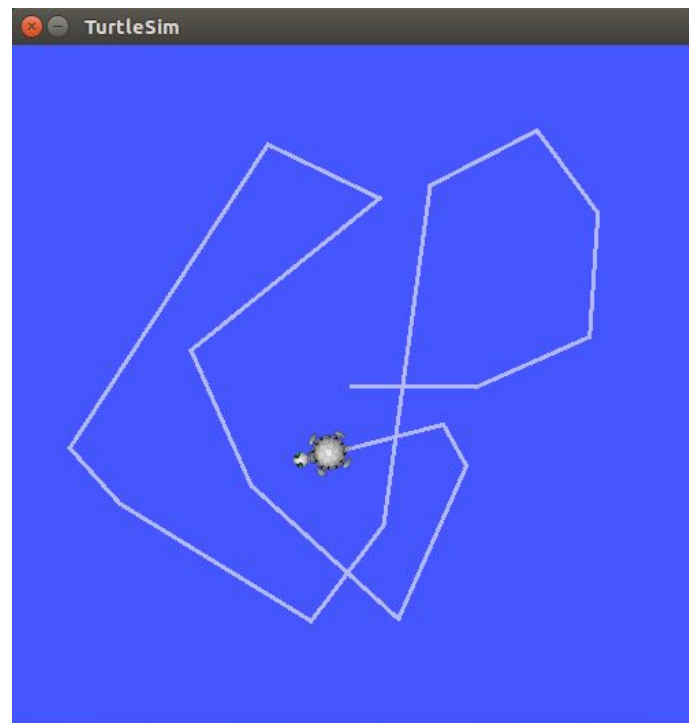
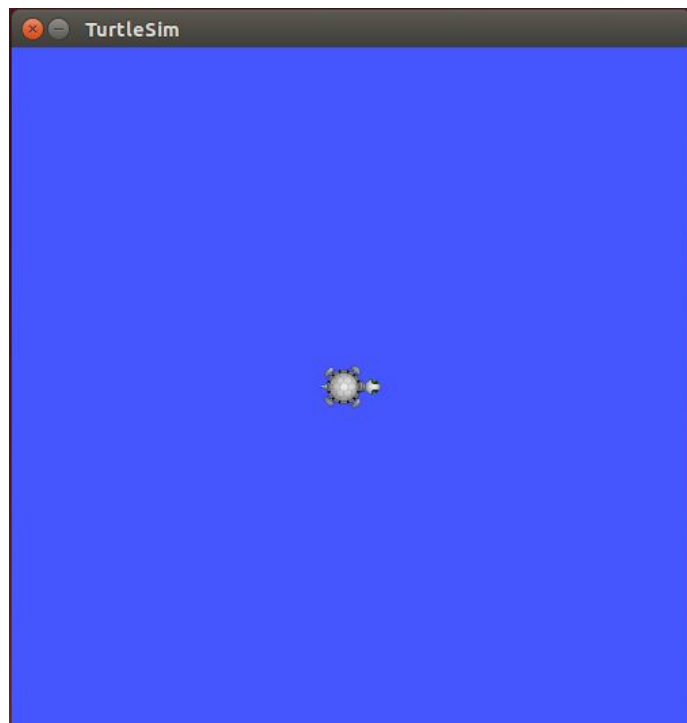
Exemplo

Usaremos um exemplo para estudar os conceitos básicos do ROS. Executar em três terminais diferentes:

```
roscore
```

```
roslaunch turtlesim turtlesim_node
```

```
roslaunch turtlesim turtlesim_teleop_key
```



Exemplo

- Esses comandos executarão o *turtlesim*, que é um simulador simples instalado junto com o ROS.
- Mantendo o terceiro terminal ativo, é possível controlar a tartaruga usando as setas do teclado.
- Esse exemplo apresenta dois nós se comunicando através da publicação de mensagens. Estudaremos esses conceitos a seguir.

Pacotes

- Todo o software no ROS é organizado em pacotes.
- Um pacote no ROS é uma coleção coerente de arquivos, geralmente incluindo tanto executáveis quanto arquivos de suporte.

Pacotes

- Listar todos os pacotes instalados:

```
rospack list
```

- Descobrir em qual pasta está instalado um pacote:

```
rospack find nome-do-pacote
```

- Exemplo:

```
rospack find turtlesim
```

Pacotes

- Todo pacote é definido por um manifesto, um arquivo chamado `package.xml`. Esse arquivo define alguns detalhes do pacote incluindo seu nome, versão, mantenedor e dependências.

- Inspeccionar a pasta de um pacote:

```
rosls nome-do-pacote
```

- Ir para a pasta do pacote:

```
roscd nome-do-pacote
```


Pacotes

Exemplo: Ver as imagens das tartarugas do turtlesim:

```
rosls turtlesim
```

```
rosls turtlesim/images
```

```
roscd turtlesim/images
```

```
eog box-turtle.png
```

ROS Master

Um dos objetivos do ROS é permitir que os roboticistas projetem software como um grupo de pequenos programas independentes uns dos outros, chamados nós, que são executados ao mesmo tempo. Para isso, os nós precisam ser capazes de se comunicar uns com os outros. O ROS Master é o programa que permite e gerencia essa comunicação.

ROS Master

- Para iniciar o master:

```
roscore
```

- O comando roscore deve ser executado no início da execução de uma aplicação do ROS e deve continuar aberto durante todo o tempo da execução.

Nós

- Um nó é uma instância de um programa que está sendo executado.
- Para iniciar um nó:

```
roslaunch nome-do-pacote nome-do-executavel
```

- No exemplo do turtlesim, iniciamos dois nós: `turtlesim_node` e `turtle_teleop_key`

Nós

- Para listar todos os nós que estão sendo executados:

```
roscout list
```

- Obs: O nó `/roscout` é um nó especial que é inicializado automaticamente pelo roscore.

Nós

- Obter informações sobre um nó:

```
roscnode info nome-do-no
```

- Encerrar um nó:

```
roscnode kill nome-do-no
```

ou Ctrl-c no terminal onde foi iniciado.

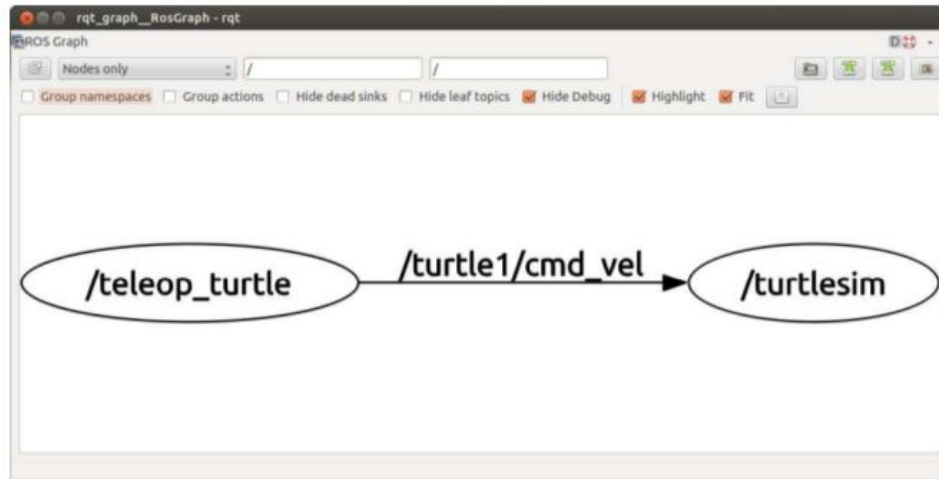
Tópicos e Mensagens

- No nosso exemplo, os nós `/turtlesim` e `/teleop_turtle` estão se comunicando de alguma forma.
- A forma mais básica que o ROS utiliza para fazer a comunicação entre os nós é enviando mensagens. As mensagens no ROS são organizadas em tópicos. A ideia é que os nós que querem compartilhar informação publicam mensagens no nó apropriado, enquanto que os nós que querem receber essa informação subscrevem naquele tópico. O ROS master garante que os nós publicadores e subscritores encontrem uns aos outros.

Tópicos e Mensagens

Visualizar uma representação gráfica dos nós e tópicos:

```
rqt_graph
```



Mensagens e Tipos de Mensagens

- Listar tópicos:

```
rostopic list
```

- Imprimir as mensagens de um tópico:

```
rostopic echo nome-do-topico
```

- Exemplo:

```
rostopic echo /turtle1/cmd_vel
```

Mensagens e Tipos de Mensagens

- Obter informações sobre um tópico:

```
rostopic info nome-do-topico
```

- Exemplo:

```
rostopic info /turtle1/color_sensor
```

Mensagens e Tipos de Mensagens

- Obter informações sobre um tipo de mensagem:

```
rosmg show nome-do-tipo-de-mensagem
```

- Exemplos:

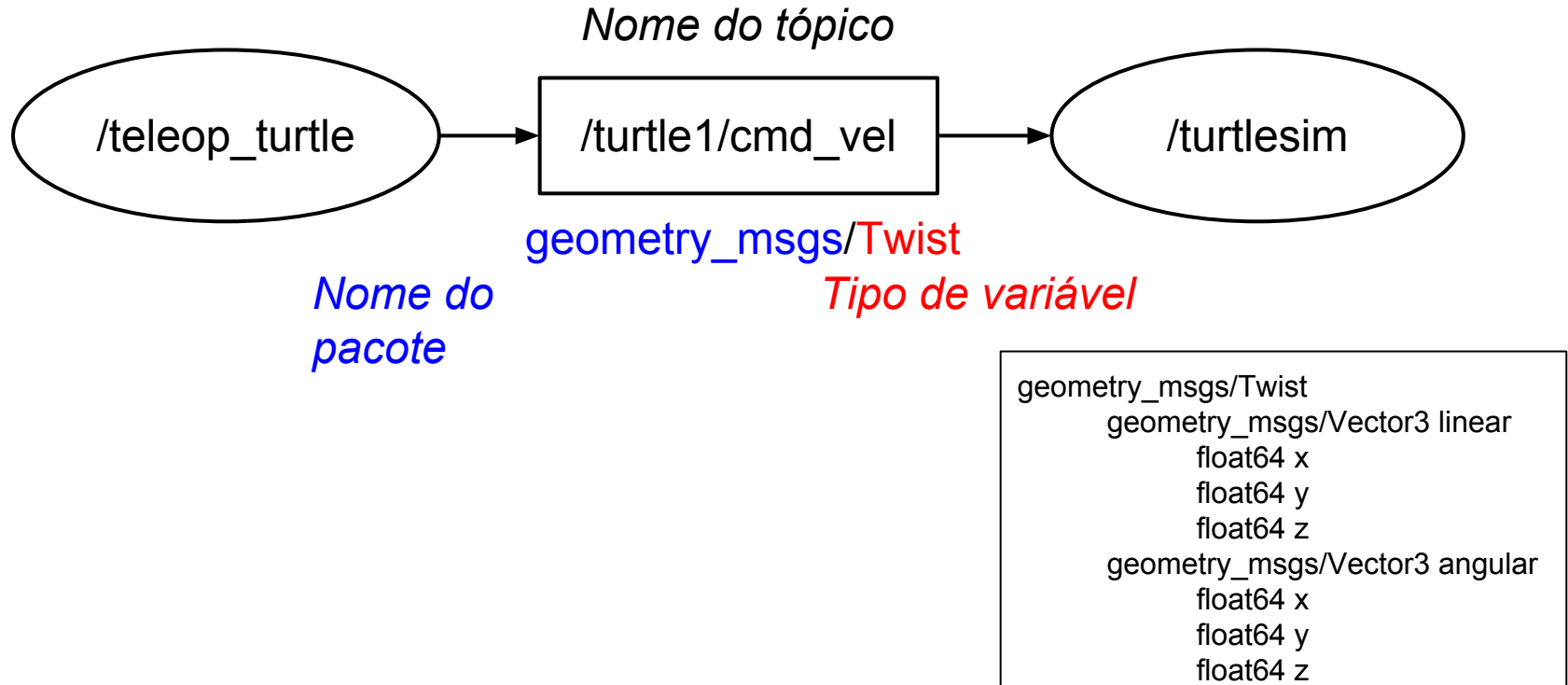
```
rosmg show turtlesim/Color
```

```
rosmg show geometry_msgs/Twist
```

- Para ver mais detalhes sobre a mensagem, usar a opção -r:

```
rosmg show -r nome-do-tipo-de-mensagem
```

Mensagens e Tipos de Mensagens



Mensagens e Tipos de Mensagens

- Publicar mensagens pela linha de comando:

```
rostopic pub nome-do-topico tipo-da-mensagem "conteudo-da-mensagem"
```

- Exemplo

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

Um exemplo maior

```
roslaunch turtlesim turtlesim_node __name:=A
```

```
roslaunch turtlesim turtlesim_node __name:=B
```

```
roslaunch turtlesim turtle_teleop_key __name:=C
```

```
roslaunch turtlesim turtle_teleop_key __name:=D
```

- O que vai aparecer no rqt_graph?

Criar um Workspace

Antes de começarmos a criar nossos próprios pacotes é necessário criar um workspace, que é uma pasta onde todos os nossos pacotes ficarão.

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

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

```
catkin_init_workspace
```

```
cd ~/catkin_ws
```

```
catkin_make
```

Criar um Workspace

Para tornar os pacotes dentro do nosso workspace visíveis para o sistema do ROS, executar os comandos:

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

```
source ~/.bashrc
```


Criando nosso primeiro pacote

- O comando para criar um pacote é:

```
catkin_create_pkg nome-do-pacote
```

- Criar um pacote para ser utilizado no curso:

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

```
catkin_create_pkg ros_e_gazebo
```

Criando nosso primeiro pacote

- `package.xml`: é o manifesto, que já foi explicado anteriormente
- `CMakeLists.txt`: é um script que será utilizado pelo catkin para construir os arquivos do projeto. Contém instruções como quais executáveis serão criados, quais arquivos fonte utilizar para criá-los e onde encontrar as bibliotecas que devem ser importadas.

Criando nosso primeiro pacote

No arquivo `package.xml`:

- A maioria dos campos é auto explicativa;
- `<build_depend>` e `<run_depend>` - Aqui são listadas as dependências do pacote. Editar o arquivo, adicionando `roscpp`, `geometry_msgs` e `turtlesim` como dependências.

Criando nosso primeiro pacote

```
<build_depend>roscpp</build_depend>  
<build_depend>geometry_msgs</build_depend>  
<build_depend>turtlesim</build_depend>  
  
<run_depend>roscpp</run_depend>  
<run_depend>geometry_msgs</run_depend>  
<run_depend>turtlesim</run_depend>
```

Criando nosso primeiro pacote

No arquivo CmakeLists.xml:

- `project(ros_e_gazebo)` – nome do pacote
- `find_package(catkin REQUIRED)` – lista as dependencias do pacote. Editar essa linha deixando da seguinte forma:

```
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs  
turtlesim)
```
- `catkin_package()` - Declara um pacote catkin

Criando nosso primeiro pacote

Compilar o pacote criado:

```
cd ~/catkin_ws
```

```
catkin_make
```

Criando nosso primeiro programa

hello.cpp

na pasta src

```
1  // This is a ROS version of the standard "hello , world"
2  // program.
3
4  // This header defines the standard ROS classes.
5  #include <ros/ros.h>
6
7  int main(int argc , char **argv) {
8      // Initialize the ROS system.
9      ros::init(argc , argv , "hello_ros");
10
11     // Establish this program as a ROS node.
12     ros::NodeHandle nh;
13
14     // Send some output as a log message.
15     ROS_INFO_STREAM("Hello , _ROS! ");
16 }
```

Criando nosso primeiro programa

- `#include <ros/ros.h>` - Inclui as classes padrão do ROS;
- `ros::init(argc, argv, "hello_ros");` - Inicia o sistema do ROS, declarando um nó chamado "hello_ros";
- `ros::NodeHandle nh;` - Cria um objeto NodeHandle para acessar as funções do ROS;
- `ROS_INFO_STREAM("Hello ROS!");` - Imprime a mensagem na tela;

Criando nosso primeiro programa

Editar o arquivo CMakeLists.txt:

Declarar executáveis:

```
add_executable(nome-do-executavel arquivos-fonte)
target_link_libraries(nome-do-executavel ${catkin_LIBRARIES})
```

No nosso caso:

```
add_executable(hello hello.cpp)
target_link_libraries(hello ${catkin_LIBRARIES})
```

Criando nosso primeiro programa

- Compilar o pacote

```
cd ~/catkin_ws  
catkin_make
```

Criando nosso primeiro programa

Executar o programa

```
roscore
```

```
roslaunch ros_e_gazebo hello
```



Aula 2: Publishers, Subscribers e Launch



Criando um Publisher

Agora nós vamos criar um programa que publica mensagens de comando de velocidade aleatórias para o turtlesim.

Criando um Publisher

pubvel.cpp

```
1 // This program publishes randomly-generated velocity
2 // messages for turtlesim.
3 #include <ros/ros.h>
4 #include <geometry_msgs/Twist.h> // For geometry_msgs::Twist
5 #include <stdlib.h> // For rand() and RAND_MAX
6
7 int main(int argc, char **argv) {
8     // Initialize the ROS system and become a node.
9     ros::init(argc, argv, "publish_velocity");
10    ros::NodeHandle nh;
11
12    // Create a publisher object.
13    ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
14        "turtle1/cmd_vel", 1000);
15
16    // Seed the random number generator.
17    srand(time(0));
18
```

Criando um Publisher

```
19  // Loop at 2Hz until the node is shut down.
20  ros::Rate rate(2);
21  while(ros::ok()) {
22      // Create and fill in the message. The other four
23      // fields, which are ignored by turtlesim, default to 0.
24      geometry_msgs::Twist msg;
25      msg.linear.x = double(rand())/double(RAND_MAX);
26      msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
27
28      // Publish the message.
29      pub.publish(msg);
30
31      // Send a message to rosout with the details.
32      ROS_INFO_STREAM("Sending random velocity command: "
33          << "linear=" << msg.linear.x
34          << "angular=" << msg.angular.z);
35
36      // Wait until it's time for another iteration.
37      rate.sleep();
38  }
39 }
```

Criando um Publisher

Incluir os arquivos de cabeçalho necessários

- `#include <geometry_msgs/Twist.h>` - contém a classe necessária para criar mensagens do tipo que precisamos;
- `#include <stdlib.h>` - para usar `rand()` e `RAND_MAX`

Criando um Publisher

Criar um objeto da classe `ros::Publisher`

```
ros::Publisher nome-do-objeto = node-handle.advertise<tipo-da-mensagem>  
    (nome-do-topico, tamanho-da-fila);
```

- *nome-do-objeto*: Usar um nome que faça sentido, como `cmdVelPub` ou apenas `pub` caso só exista um Publisher;
- *node-handle*: Objeto da classe `ros::NodeHandle` criado previamente;
- *tipo-da-mensagem*: Nome da classe do tipo de mensagem que será publicado;

Criando um Publisher

- *nome-do-topico*: Escolher um nome que faça sentido. No nosso caso vamos publicar em um tópico específico que foi criado pelo turtlesim;
- *tamanho-da-fila*: Caso mensagens estejam sendo publicadas mais rápido do que consumidas, o ROS vai guardar essas mensagens em uma fila. Usar um número grande como 1000 geralmente evita qualquer problema.

Criando um Publisher

Selecionar uma semente para o gerador de números aleatórios

```
srand(time(0));
```

Criando um Publisher

Criar e preencher a mensagem

```
geometry_msgs::Twist msg;  
msg.linear.x = double(rand())/double(RAND_MAX);  
msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
```

Esse código preenche os campos velocidade linear com um valor entre 0 e 1 e velocidade angular com um número entre -1 e 1. O turtlesim ignora os outros campos.

Criando um Publisher

Publicar a mensagem:

```
pub.publish(msg);
```

Criando um Publisher

Para publicar as mensagens de forma contínua e periódica, usamos um loop while. A condição de repetição do loop é:

```
ros::ok()
```

Essa função retorna true enquanto o nosso nó estiver rodando corretamente. Ela só retornará false caso o nó seja encerrado, nos seguintes casos: o nó seja encerrado com `roscpp kill`, ou com Ctrl-C, ou chamando a função `ros::shutdown()` dentro do código, ou iniciando outro nó com o mesmo nome.

Criando um Publisher

Criar uma taxa de publicação:

```
ros::Rate rate(2);
```

E dentro do loop, chamar a função:

```
rate.sleep();
```

Isso vai fazer com que o ROS espere um tempo entre cada iteração do loop. O ROS vai calcular esse tempo automaticamente de forma que o loop seja executado 2 vezes por segundo.

Criando um Publisher

Usar ROS_INFO_STREAM para imprimir os valores publicados na tela.

```
ROS_INFO_STREAM("Sending random velocity command:")  
  << " linear=" << msg.linear.x  
  << " angular=" << msg.angular.z);
```


Criando um Publisher

Para compilar o pubvel:

- Adicionar o novo executável no arquivo CmakeLists.txt.

```
cd ~/catkin_ws && catkin_make
```

Criando um Publisher

Para executar:

```
roscore  
roslaunch ros_e_gazebo pubvel  
roslaunch turtlesim turtlesim_node
```

Verificar a frequência de publicação:

```
rostopic hz /turtle1/cmd_vel
```

Criando um Subscriber

Agora nós vamos criar um programa que subscreve ao tópico `/turtle1/pose`, no qual o `turtlesim_node` publica. As mensagens nesse tópico descrevem a **pose** da tartaruga, um termo que se refere à posição e orientação.

subpose.cpp

```
1 // This program subscribes to turtle1/pose and shows its
2 // messages on the screen.
3 #include <ros/ros.h>
4 #include <turtlesim/Pose.h>
5 #include <iomanip> // for std::setprecision and std::fixed
6
7 // A callback function. Executed each time a new pose
8 // message arrives.
9 void poseMessageReceived(const turtlesim::Pose& msg) {
10     ROS_INFO_STREAM(std::setprecision(2) << std::fixed
11         << "position=(" << msg.x << ", " << msg.y << ") "
12         << "direction=" << msg.theta);
13 }
14
15 int main(int argc, char **argv) {
16     // Initialize the ROS system and become a node.
17     ros::init(argc, argv, "subscribe_to_pose");
18     ros::NodeHandle nh;
19
20     // Create a subscriber object.
21     ros::Subscriber sub = nh.subscribe("turtle1/pose", 1000,
22         &poseMessageReceived);
23
24     // Let ROS take over.
25     ros::spin();
26 }
```

Criando um Subscriber

- Uma diferença importante entre publicar e subscrever é que o Subscriber não sabe quando as mensagens vão chegar, portanto nós precisamos escrever um código que será chamado automaticamente toda vez que uma nova mensagem chegue. Esse código é chamado de uma função callback.

```
void nome-da-funcao( const nome-do-pacote::nome-do-tipo &msg ) { ... }
```

Criando um Subscriber

- O corpo da função tem acesso a todos os campos da mensagem recebida através da variável `msg`, e podemos utilizar esses dados da maneira que quisermos. No nosso caso, nós apenas imprimimos os campos da mensagem na tela.
- É necessário incluir o arquivo `turtlesim/Pose.h`
- A função callback sempre retorna `void`.

Criando um Subscriber

- Criar um objeto subscriber:

```
ros::Subscriber nome-do-objeto = node-handle.subscribe( nome-do-topico,  
                                                         tamanho-da-fila, ponteiro-para-funcao-callback );
```

- Para usar o ponteiro basta colocar um & antes do nome da função

Criando um Subscriber

O ROS só vai chamar a função callback quando passarmos o controle do programa para ele. Existem duas formas de fazer isso:

1ª Forma:

```
ros::spinOnce();
```

Essa forma pede para o ROS executar todos os callbacks e então retornar o controle para nós.

Criando um Subscriber

Essa forma é útil quando queremos fazer alguma coisa entre as execuções dos callbacks:

```
While (ros::ok()) {  
  
    // Fazer alguma tarefa. Por exemplo, publicar mensagens  
  
    ros::spinOnce();  
  
}
```

Criando um Subscriber

2ª Forma:

```
ros::spin();
```

Essa forma diz para o ROS continuar executando os callbacks sempre que necessário indefinidamente, até que o nó seja encerrado.

Criando um Subscriber

Para compilar:

- Adicionar o novo executável no arquivo `CmakeLists.txt`.
- `catkin_make`

Criando um Subscriber

Para executar:

```
roscore
```

```
roslaunch turtlesim turtlesim_node
```

```
roslaunch ros_e_gazebo subpose
```

```
roslaunch ros_e_gazebo pubvel
```

Exercício

- Usando os conceitos que estudamos e os programas que escrevemos, crie um nó que move a tartaruga do turtlesim para uma posição (X, Y) determinada.
- Escreva um arquivo launch que permita executar o turtlesim e o nó que foi criado com apenas um comando.

Exercício 2

- Modifique o programa do Exercício anterior para subscrever a um tópico chamado /goal.
- Deverá ser possível, durante a execução do programa, publicar mensagens representando posições no tópico /goal.
- A tartaruga deverá tratar essa posição como seu novo objetivo e se mover para lá.

Arquivos Launch

Arquivos Launch

Esses arquivos nos permitem executar vários nós ao mesmo tempo. A ideia é listar todos os nós que queremos executar em uma sintaxe xml específica, podendo definir configurações para cada nó e passar argumentos.

Arquivos Launch

example.launch:

```
<launch>
```

```
  <node pkg="turtlesim" type="turtlesim_node" name="turtlesim" respawn="true" />
```

```
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop_key" required="true"  
    launch-prefix="xterm -e" />
```

```
  <node pkg="ros_e_gazebo" type="subpose" name="pose_subscriber" output="screen" />
```

```
</launch>
```

Arquivos Launch

Para executar um arquivo launch:

```
roslaunch nome_do_pacote arquivo.launch
```

Esse arquivo executa todos os nós do exemplo anterior, mas com um único comando:

```
roslaunch ros_e_gazebo exemplo.launch
```

Criando Arquivos Launch

- Tudo deve estar envolvido em uma tag launch: `<launch> ... </launch>`
- Cada nó é chamado por uma tag node:

```
<node pkg="pacote" type="executavel" name="nome-do-no" />
```

- O atributo `name` sobrescreve o nome definido no código do nó
- O atributo `output="screen"` serve para que a saída do nó seja impressa na tela.
- O atributo `launch-prefix` permite adicionar um comando a ser executado antes do nó ser lançado. Neste caso a ideia era criar um terminal separado para o `teleop_turtle`.

Criando Arquivos Launch

É possível incluir outros arquivos launch:

```
<include file="caminho-para-o-arquivo-launch" />
```

Criando Arquivos Launch

É possível incluir argumentos:

```
<arg name="nome-do-argumento" default="valor-padrao" />
```

Essa tag define um argumento, que pode ser passado pelo comando `roslaunch` e utilizado em qualquer lugar do arquivo através da sintaxe:

```
$(arg nome-do-argumento)
```

Arquivos Launch

Por exemplo:

```
<launch>
```

```
  <arg name="node_name" default="hello" />
```

```
  <node pkg="simuladores" type="hello" name="$(arg node_name)" />
```

```
</launch>
```

Arquivos Launch

Chamar esse arquivo com o comando:

```
roslaunch simuladores hello.launch node_name:=hi
```

Sobrescrevemos o valor padrão do argumento `node_name` que era "hello" com o novo valor "hi".

Gazebo Simulator

Gazebo

O Gazebo é um simulador 3D que tem a habilidade de simular, de forma precisa e eficiente, populações de robôs em ambientes indoor e outdoor complexos. Ele já vem com uma base contendo diversos modelos de objetos, robôs e sensores, mas permite também que criemos nossos próprios ambientes e modelos. É capaz de simular vários tipos de sensores como sonar, lidar, GPS e câmera.

Instalação do Gazebo

- Se, ao instalarmos o ROS, escolhermos o pacote `ros-kinetic-desktop-full`, o Gazebo 7 já vem instalado. Essa é a versão indicada para o ROS Kinetic e é a que vamos utilizar no curso.
- Para instalar o Gazebo de forma independente, e também numa versão mais nova, basta seguir as instruções na página a seguir:
http://gazebosim.org/tutorials?tut=install_ubuntu&cat=install

Iniciar o Gazebo

- Para iniciar o Gazebo, basta abrir um terminal e executar o comando:

```
gazebo
```

- Isso inicializará o Gazebo como um programa independente do ROS.
- Para utilizar o Gazebo juntamente com o ROS, é necessário o pacote `gazebo_ros`.
- Caso o pacote ainda não esteja instalado:

```
sudo apt-get install ros-kinetic-gazebo-ros
```

Iniciar o Gazebo

- Para iniciar o Gazebo como parte do ROS, executar os seguintes comandos em dois terminais independentes:

```
roscore
```

```
roslaunch gazebo_ros gazebo
```

- Dessa vez o Gazebo será iniciado como um nó do ROS, capaz de publicar e subscrever em tópicos.

```
rostopic list
```



Componentes do Gazebo



Mundos

O mundo que será simulado pelo Gazebo pode conter diversos objetos, robôs e sensores. Diversas características podem ser alteradas, como vento, luminosidade e mesmo as regras da física. Os mundos são descritos em arquivos com extensão `.world`, que são escritos numa linguagem de marcação chamada SDF (Simulation Description Format).

Mundos

Exemplo: `empty.world`

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <!-- Global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
  </world>
</sdf>
```

Mundos

Outros exemplos de mundos:

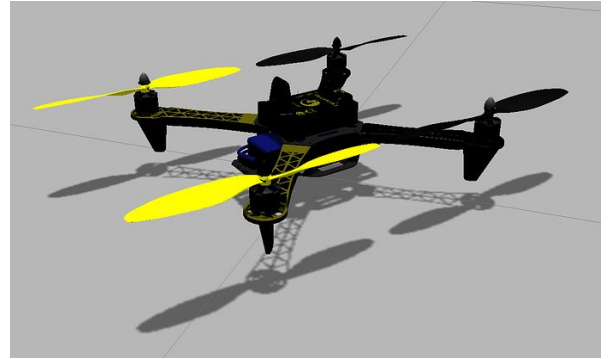
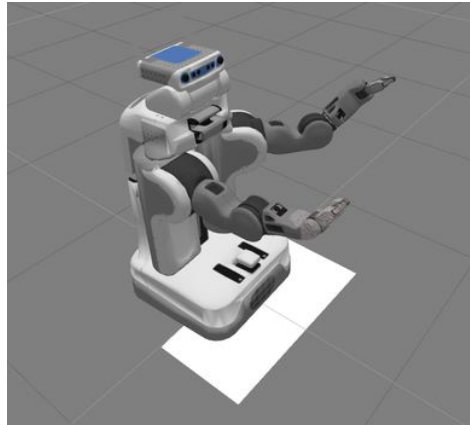
```
roslaunch gazebo_ros willowgarage_world.launch
```

```
roslaunch gazebo_ros shapes_world.launch
```

```
roslaunch gazebo_ros rubble_world.launch
```


Modelos

Modelos representam elementos da simulação: objetos, sensores ou mesmo robôs. Modelos são descritos em arquivos com a extensão `.sdf`, e devem contar uma única tag `<model> ... </model>`. São escritos usando a mesma linguagem SDF dos arquivos `world`.



Modelos de Robôs no ROS

- O ROS também utiliza arquivos para representar modelos de robôs, porém utiliza uma linguagem diferente, a URDF - Universal Robotic Description Format.
- Essa linguagem é muito parecida com a SDF do Gazebo, porém mais limitada: ela só pode ser usada para representar robôs, e não objetos estáticos, e não possui alguns elementos exclusivos de simulação.
- Quando o Gazebo encontra um arquivo URDF, ele primeiro converte para SDF e só então carrega aquele modelo no ambiente de simulação.

Plugins

- Plugins são programas que nos permitem interagir com o ambiente de simulação do Gazebo.
- Através de plugins é possível controlar robôs, simular sensores, modificar as leis da física, criar novos robôs ou objetos, etc.
- Plugins são escritos em C++.



Construindo nossa própria simulação

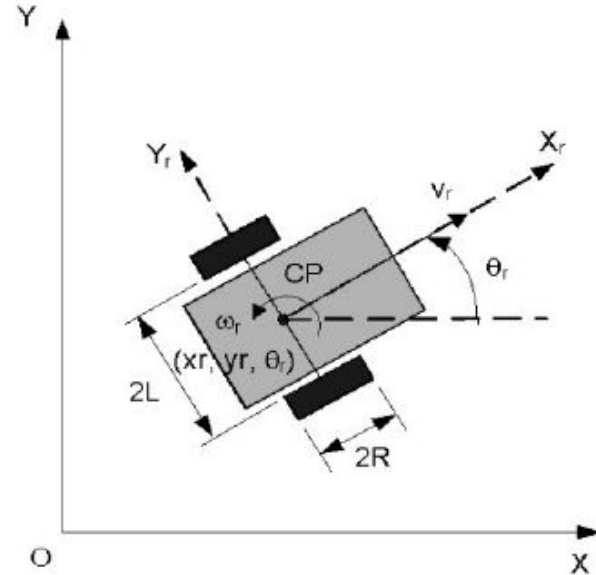
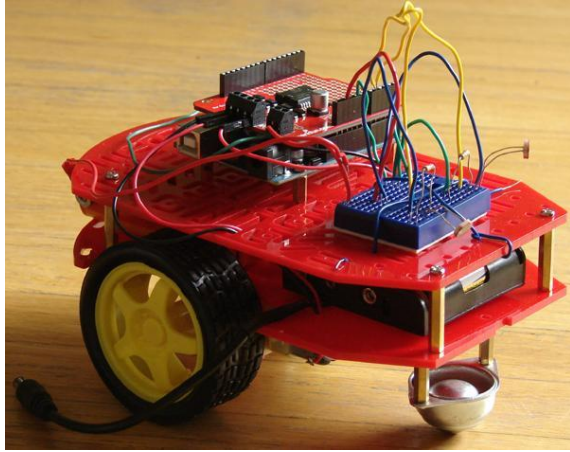


Construindo nossa simulação

- Para compreendermos melhor como utilizar os componentes do Gazebo, vamos criar alguns pacotes que definem uma simulação e executar no Gazebo.
- Vamos criar os arquivos que definem um robô de direção diferencial. Esse tipo de robô possui duas rodas motorizadas que se movem de forma independente e uma caster wheel, uma roda que se move livremente e serve para sustentar o robô.

Construindo nossa simulação

- Como resultado o robô é capaz de se mover para frente e para trás, e também de girar em torno do eixo Z. São movimentos muito similares aos da tartaruga no turtlesim.



Construindo nossa simulação

- Primeiro, vamos criar três novos pacotes:

`mybot_gazebo`

`mybot_description`

`mybot_control`

Construindo nossa simulação

Vamos usar como base os modelos e arquivos de configurações deste repositório:

```
cd ~
```

```
git clone -b base https://github.com/lvribeiro/mybot\_ws.git
```

Após clonar o repositório, ir para a pasta raiz do workspace e compilar:

```
cd ~/mybot_ws
```

```
catkin_make
```


Organização dos diretórios

- mybot_control
- mybot_description

Pacote com os nós que controlaram o robo

Pacote com descrição do modelo do robo

- urdf

- macros.xacro
- materials.xacro
- mybot.gazebo
- mybot.xacro

Arquivo com definição dos plugins do gazebo

Arquivo principal do modelo

- mybot_gazebo

Pacote com definição do mundo e arquivos de inicialização

- launch

- mybot_world.launch

Inicia Gazebo com o robo definido em mybot.xacro

- worlds

- mybot.world

Mundo para simulação

Mundo

Mybot.world

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://sun</uri>
    </include>
  </world>
</sdf>
```

Mundo

- Esse é um arquivo `.world` bem básico, que apenas define o nome do mundo e inclui dois modelos: um piso no chão e o sol para fornecer iluminação.
- Vamos criar um arquivo `.launch` que inicializará o Gazebo como um nó do ROS e carrega o mundo de simulação que criamos.

```
roscd mybot_gazebo/launch
```

```
gedit mybot_world.launch
```

Mundo

- Launch file para inicializar o mundo: `mybot_world.launch`

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <arg name="gui" default="true"/>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find mybot_gazebo)/worlds/mybot.world"/>
    <arg name="gui" value="$(arg gui)"/>
  </include>
</launch>
```

Mundo

- Agora vamos executar o arquivo `.launch`:

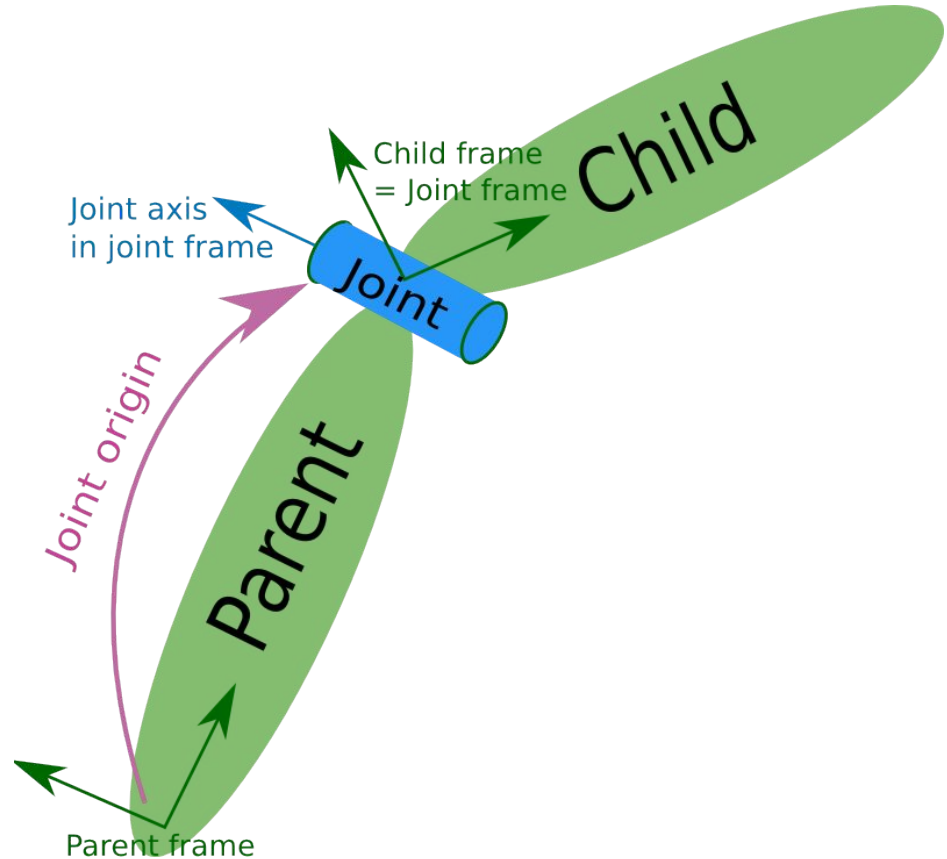
```
roslaunch mybot_gazebo mybot_world.launch
```

Modelo

- Um modelo de um robô é formado por links e joints.
- Links são as partes do robô: o corpo, cada roda, eventuais sensores, etc.
- Joints são as ligações entre os links. Existem diversos tipos de joints, e cada tipo define como o link é capaz de se mover em relação ao outro link ao qual está ligado.
- Usamos as tags xml para definir os diversos elementos que formam o robô.

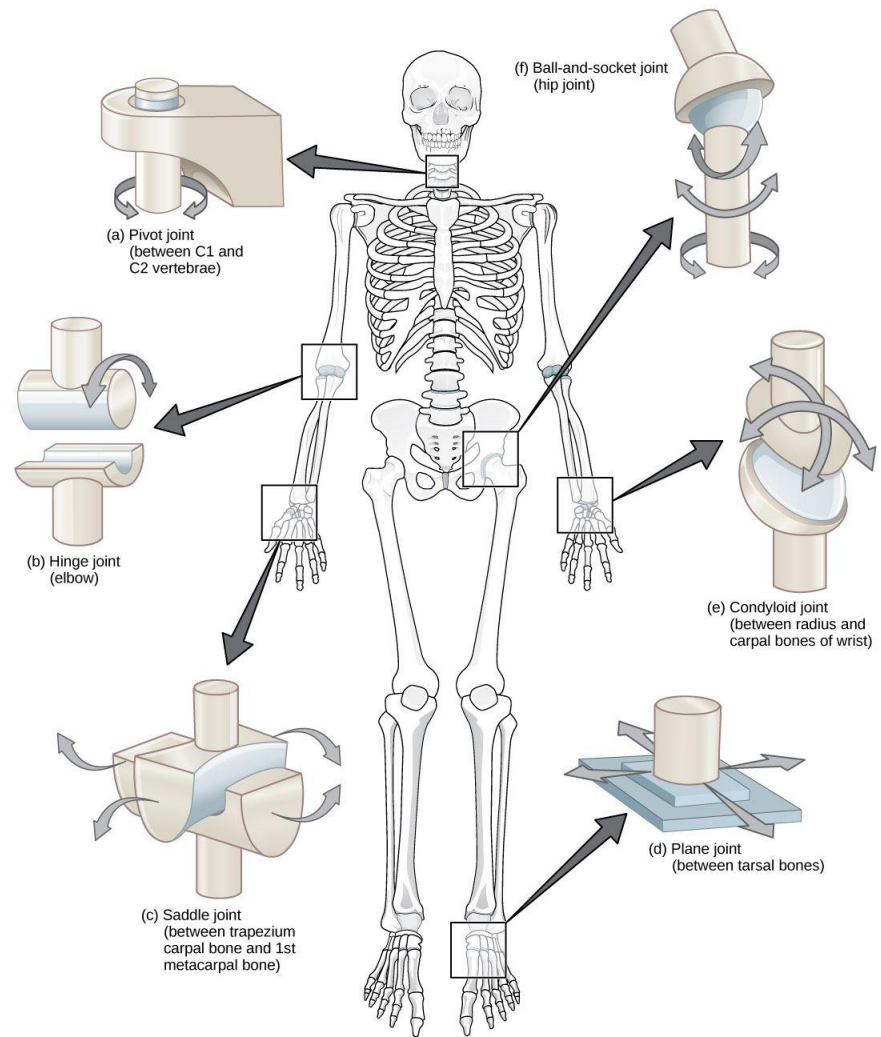
Modelo

Diagrama de dois
Links conectados
por um *Joint*



Tipos de Joints

- revolute
- continuous
- prismatic
- fixed
- floating
- planar



Controlando o Robô

- Para controlar o robô, vamos utilizar um plugin fornecido pelo Gazebo.
- Vamos adicionar mais um código no arquivo URDF, instruindo o Gazebo a carregar o plugin e passando as configurações necessárias.
- Adicione no final do arquivo URDF o código a seguir.

```
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
    <legacyMode>false</legacyMode>
    <alwaysOn>true</alwaysOn>
    <updateRate>20</updateRate>
    <leftJoint>left_wheel_hinge</leftJoint>
    <rightJoint>right_wheel_hinge</rightJoint>
    <wheelSeparation>0.4</wheelSeparation>
    <wheelDiameter>0.1</wheelDiameter>
    <torque>20</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>chassis</robotBaseFrame>
  </plugin>
</gazebo>
```

Controlando o Robô

- Esse plugin subscreve no tópico `/cmd_vel` e aguarda mensagens de comando de velocidade de forma similar à tartaruga do turtlesim.
- E assim como o turtlesim, ele publica a posição do robô em um tópico chamado `/odom`

Adicionando Sensores

Adicionando sensores

- Para que o robô possa sentir o ambiente ao seu redor, precisamos adicionar sensores a ele.
- Os sensores publicam sua informação em tópicos do ROS.
- Existe um tipo de mensagem específica para cada tipo de sensor, por exemplo `sensor_msgs/Image` para câmeras, `sensor_msgs/LaserScan` para lasers, `sensor_msgs/NavSatFix` para GPS, etc.
- No Gazebo os sensores são implementados através de plugins.

Adicionando sensores

- Para adicionar um sensor ao nosso modelo, é necessário incluir três novos elementos:
- Um link, que representa o corpo físico do sensor;
- Uma joint, ligando o sensor ao corpo do robô;
- Um plugin, que implementa o funcionamento do sensor.
- O Gazebo fornece plugins para diversos tipos de sensores, como lasers, lidars, câmeras, IMU, entre outros.

Laser

- Vamos adicionar um laser ao nosso robô;
- Esse é um tipo de sensor de distância que emite diversos feixes de luz e mede quanto tempo a luz demora para ir até um obstáculo e voltar. Assim ele é capaz de calcular qual a distância até aquele ponto.
- Vamos adicionar um sensor que emite 8 lasers espalhados em todas as direções.

Adicionar link e joint em
mybot.xacro

```
<joint name="hokuyo_joint" type="fixed">  
  <axis xyz="0 1 0" />  
  <origin xyz=".15 0 .1" rpy="0 0 0"/>  
  <parent link="chassis"/>  
  <child link="hokuyo"/>  
</joint>
```



```
<link name="hokuyo">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
<box size="0.1 0.1 0.1"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://mybot_description/meshes/hokuyo.dae"/>
    </geometry>
  </visual>
  <inertial>
    <mass value="1e-5" />
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
  </inertial>
</link>
```

Adicionar link e joint em
mybot.xacro

```
<gazebo reference="hokuyo">
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.570796</min_angle>
          <max_angle>1.570796</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.10</min>
        <max>30.0</max>
        <resolution>0.01</resolution>
      </range>
    </ray>
  </sensor>
</gazebo>
```

Adicionar plugin em
mybot.gazebo

Adicionar plugin em
mybot.gazebo

```
<noise>
  <type>gaussian</type>
  <mean>0.0</mean>
  <stddev>0.01</stddev>
</noise>
</ray>
<plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
  <topicName>/mybot/laser/scan</topicName>
  <frameName>hokuyo</frameName>
</plugin>
</sensor>
</gazebo>
```

Laser

- Esse sensor que acabamos de adicionar publica mensagens do tipo `sensor_msgs/LaserScan` no tópico `mybot/laser/scan` (o nome do tópico pode ser configurado no código do sensor).
- Essa mensagem possui um campo chamado `ranges`, que é um array contendo as medidas de cada um dos lasers.
- Vamos agora escrever um nó que é capaz de ler essas mensagens.
- Na pasta `src` do pacote `mybot_control`, crie um arquivo chamado `scansub.cpp`.

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
#include <sstream>

void laserScanCallback(const sensor_msgs::LaserScan& msg)
{
    std::ostringstream oss;
    oss << "Ranges = [ ";

    for (int i = 0; i < msg.ranges.size(); i++)
    {
        oss << msg.ranges[i] << " ";
    }

    oss << " ];";

    ROS_INFO_STREAM( oss.str() );
}
```

```
int main(int argc, char** argv)
{

    ros::init(argc, argv, "laser_scan_sub");
    ros::NodeHandle nh;

    ros::Subscriber scan_sub = nh.subscribe("mybot/laser/scan",
        1000, &laserScanCallback);

    ros::spin();
}
```

Laser

- Esse programa é muito parecido com o subpose, porém ao invés de usarmos mensagens do tipo `turtlesim/Pose` estamos usando mensagens do tipo `sensor_msgs/LaserScan`.
- Dentro da função `laserScanCallback`, o objeto `msg` contém a mensagem que foi recebida. O array `msg.ranges` contém as medidas de cada um dos sensores.
- Fazemos um `for` para ler cada uma das medidas e concatenar em uma string para imprimir na tela.

Laser

- Para compilar o programa:
- Adicionar as dependências no arquivo `package.xml`

```
<build_depend>roscpp</build_depend>  
<build_depend>sensor_msgs</build_depend>  
  
<run_depend>roscpp</run_depend>  
<run_depend>sensor_msgs</run_depend>
```


Laser

- Adicionar também as dependências no arquivo `CMakeLists.txt`:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  sensor_msgs
)
```

- Adicionar o novo executável no arquivo `CMakeLists.txt`:

```
add_executable(scansub src/scansub.cpp)
target_link_libraries(scansub ${catkin_LIBRARIES})
```

Laser

- Compilar o pacote

```
cd ~/catkin_ws
```

```
catkin_make
```

- Executar:

```
roslaunch mybot_control scansub
```

Câmera

Adicionar link e joint em
mybot.xacro

- Vamos agora adicionar uma câmera.

- Adicione o seguinte código ao modelo:

```
<xacro:property name="cameraSize" value="0.05"/>
```

```
<xacro:property name="cameraMass" value="0.1"/>
```

```
<joint name="camera_joint" type="fixed">
```

```
  <axis xyz="0 1 0" />
```

```
  <origin xyz=".2 0 0" rpy="0 0 0"/>
```

```
  <parent link="chassis"/>
```

```
  <child link="camera"/>
```

```
</joint>
```

```
<link name="camera">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="${cameraSize} ${cameraSize} ${cameraSize}"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="${cameraSize} ${cameraSize} ${cameraSize}"/>
    </geometry>
    <material name="green"/>
  </visual>
  <inertial>
    <mass value="${cameraMass}" />
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
  </inertial>
</link>
```

Adicionar link e joint em
mybot.xacro

```
<gazebo reference="camera">
  <material>Gazebo/Green</material>
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
    </camera>
  </sensor>
</gazebo>
```

Adicionar plugin em
mybot.gazebo

```
<plugin name="camera_controller" filename="libgazebo_ros_camera.so">
```

```
<alwaysOn>true</alwaysOn>
```

```
<updateRate>0.0</updateRate>
```

```
<cameraName>mybot/camera1</cameraName>
```

```
<imageTopicName>image_raw</imageTopicName>
```

```
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
```

```
<frameName>camera</frameName>
```

```
<hackBaseline>0.07</hackBaseline>
```

```
<distortionK1>0.0</distortionK1>
```

```
<distortionK2>0.0</distortionK2>
```

```
<distortionK3>0.0</distortionK3>
```

```
<distortionT1>0.0</distortionT1>
```

```
<distortionT2>0.0</distortionT2>
```

```
</plugin>
```

```
</sensor>
```

```
</gazebo>
```

Adicionar plugin em
mybot.gazebo

Câmera

- A câmera publica mensagens do tipo `sensor_msgs/Image` no tópico `mybot/camera1/image_raw`.
- Para visualizar as imagens, vamos utilizar o nó `image_view` do pacote `image_view`:

```
roslaunch image_view image_view image:=/mybot/camera1/image_raw
```

Câmera

- Agora, vamos escrever um nó que lê as imagens publicadas pela câmera frontal do quadrotor.
- Para isso será necessário utilizar o pacote `image_transport`. Esse é um pacote do ROS que contém diversas funções e estruturas para subscrever e publicar imagens. Ele também permite comprimir as imagens para tornar a transmissão mais rápida.
- Usaremos também algumas funções do `opencv` para visualizar as imagens. O pacote `cv_bridge` será utilizado para converter entre os formatos de imagem do `opencv` e do ROS.
- Na pasta `src` do pacote `ros_e_gazebo`, crie um arquivo chamado `camerasub.cpp`.


```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    try
    {
        cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")->image);
        cv::waitKey(30);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("Could not convert from '%s' to 'bgr8'.",
msg->encoding.c_str());
    }
}
```

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "camerasub");
    ros::NodeHandle nh;
    cv::namedWindow("view");
    cv::startWindowThread();
    image_transport::ImageTransport it(nh);
    image_transport::Subscriber sub = it.subscribe(
        "/quadrotor_1/front/image_raw", 1, imageCallback);
    ros::spin();
    cv::destroyWindow("view");
}
```

Câmera

- Primeiro, incluímos os pacotes necessários.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
```

Câmera

- Criamos uma função callback para processar as imagens recebidas:

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
```

- Dentro da função callback, utilizaremos a função `cv::imshow` do `opencv` para mostrar a imagem em uma janela:

```
try
{
    cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")->image);
    cv::waitKey(30);
}
```

Câmera

- O código:

```
cv_bridge::toCvShare(msg, "bgr8")->image
```

Tenta converter a mensagem recebida (no formato do ROS) para o formato que o opencv utiliza.

- Caso a conversão não funcione, uma mensagem de erro é mostrada:

```
catch (cv_bridge::Exception& e){  
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.",  
        msg->encoding.c_str());  
}
```

Câmera

- A função main inicia ROS e cria uma janela na qual será mostrada a imagem

```
ros::init(argc, argv, "camerasub");  
ros::NodeHandle nh;  
cv::namedWindow("view");  
cv::startWindowThread();
```

Câmera

- Para subscrever ao tópico da câmera um `NodeHandle` e um `Subscriber` convencionais não funcionariam. O pacote `image_transport` disponibiliza objetos análogos, mas que são preparados para funcionar com imagens. Note que o `Subscriber` que definimos é do pacote `image_transport`, não do `ros`:

```
image_transport::ImageTransport it(nh);  
image_transport::Subscriber sub = it.subscribe(  
    "/quadrotor_1/front/image_raw", 1, imageCallback);
```

Câmera

- Para finalizar, o controle é passado para o ROS:

```
ros::spin();
```

- Quando o programa for encerrado, é necessário destruir a janela que foi criada pelo opencv:

```
cv::destroyWindow("view");
```


Câmera

- Para compilar o programa:
- Adicionar as dependências no arquivo `package.xml`

```
<build_depend>image_transport</build_depend>  
<build_depend>cv_bridge</build_depend>
```

```
<run_depend>image_transport</run_depend>  
<run_depend>cv_bridge</run_depend>
```

Câmera

- Adicionar também as dependências no arquivo `CMakeLists.txt`:

```
find_package(catkin REQUIRED COMPONENTS
  image_transport
  cv_bridge
)
```

- Adicionar o novo executável no arquivo `CMakeLists.txt`:

```
add_executable(camerasub src/camerasub.cpp)
target_link_libraries(camerasub ${catkin_LIBRARIES})
```

Câmera

- Compilar o pacote

```
cd ~/catkin_ws
```

```
catkin_make
```

- Executar:

```
roslaunch ros_e_gazebo camerasub
```