

Robotics Lab 2023-2024

Prof. Mario Selvaggio
mario.selvaggio@unina.it

PRISMA Lab
Department of Electrical Engineering and Information Technology
University of Naples Federico II

www.wpage.unina.it/mario.selvaggio
www.prisma.unina.it

Introduction

Course description

Setup your PC

Programming for robotics

ROS - Introduction

ROS - Programming

ROS - Tools

ROS - Simulation

ROS - Sensors & controllers

ROS - Motion planning

Motion control of robotic manipulators

ROS - Kinematics and dynamic control

Robotic vision

ROS - Vision sensors

ROS - Computer vision

ROS - Visual Servoing

Control of mobile robots

Autonomous navigation and path
planning

Introduction

- **Aim of the course:** introduce students to build robotic applications quickly and efficiently using the Robot Operating System (ROS)
- **Prerequisites:** there are no formal prerequisites. Basic knowledge of the following tools can make the life a bit easier

- Linux operating system (Ubuntu)
- Software versioning (Git)
- Programming languages (C/C++)
- Robot operating System (ROS)



■ Student learning outcomes

- Understanding the ROS architecture and tools
- Creating ROS C++ programs using external libraries
- Simulating and controlling a robotic system

■ Job opportunities for robotics engineers

- Check this video



■ Evaluation

- Homeworks will be scheduled and evaluated on course days. You have to be physically present to hand them in
- The final project is to be presented in the form of a report on the day of the exam
- The work may be carried out individually or in a group of maximum 3-4 people. In the latter case, the work carried out by each member of the group must be clearly evidenced
- Video of some previous years projects (2021, 2022, 2023)

■ Grading

- Homeworks: 60%
- Project: 40%

This course requires you set up your PC. You'll find instructions on how to install and get familiar with the following computer software tools in a separate document

- **Linux**

To develop ROS applications, you will need to work in a Ubuntu environment. Install it from <https://ubuntu.com/>

- **ROS**

Main tool used to develop and running robotics applications. You need to install ROS and follow the preliminary tutorials on <https://www.ros.org/>

- **Version control software (VCS)**

Git is a VCS used to share code and keep track of it. Create an account on <https://github.com/> to hosts your Git repositories

- **Docker**

Tool for creating and managing application containers to run your applications anywhere. Visit <https://www.docker.com/> for more information

- **Computer programming**

It is highly recommended to refresh your C++ skills using any C++ tutorial, e.g. <https://www.learncpp.com/>

Programming for robotics

History of ROS

- Originally developed in 2007 at the Willow Garage and Stanford Artificial Intelligence Laboratory under GPL license
- The goal was to establish a standard way to program robots while offering off-the-shelf software components easily integrable in custom robotic applications
- Since 2013 managed by Open-Source Robotics Foundation, now Open Robotics¹
- Today used by many robots, universities and companies
- De facto standard for robot programming

¹<https://www.openrobotics.org/>

ROS main features

- Code sharing and reuse (do not reinvent the wheel)
- Distributed, modular design (nodes grouped in packages, scalable)
- Language independent (C++, Python, Java, ...)
- Individual programs communicate over defined API (ROS messages, services, etc.)
- Easy testing (ready-to-use)
- Vibrant community & collaborative environment²

²<https://robotics.stackexchange.com/>

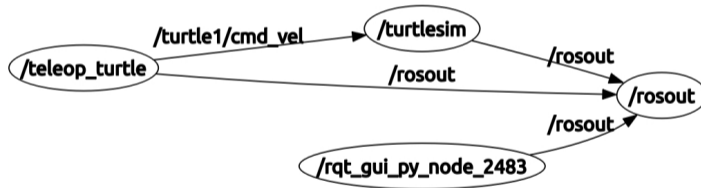
What is ROS?

- ROS is a set of software libraries and tools that help you build robot applications
- From drivers to state-of-the-art algorithms, to user interfaces, ROS provides powerful developer tools that allow you to focus on the development of your robot application



Plumbing

- ROS processes are represented as nodes in a graph structure, connected by edges called topics



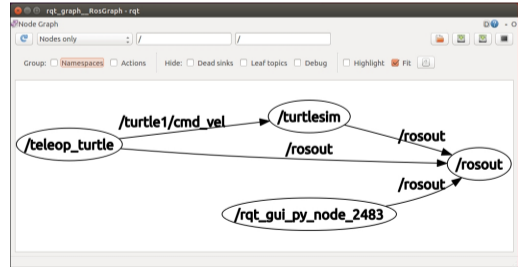
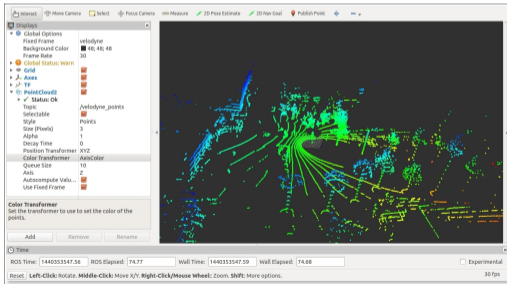
- ROS nodes** can pass messages to one another through topics, make service calls to other nodes, provide a service for other nodes, or set or retrieve shared data from a communal database called the parameter server

Plumbing (Cont'd)

- A process called the **ROS Master** makes all of this possible by registering nodes to itself, setting up node-to-node communication for topics, and controlling parameter server updates
- **Messages** and **service** calls do not pass through the master, rather the master sets up peer-to-peer communication between all node processes after they register themselves with the master
- This decentralized architecture lends itself well to robots, which often consist of a subset of networked computer hardware, and may communicate with off-board computers for heavy computing or commands

Tools

- ROS provides an extensive set of tools to configure, manage, debug, visualize, data log, and test your application



Capabilities

- ROS provides a broad collection of robot-agnostic libraries organized in packages that implement useful robot functionalities such as
 - the device driver for your GPS
 - a walk and balance controller for your quadruped
 - a mapping system for your mobile robot
- External libraries like OpenCv, PCL, and so on, are integrated in ROS thanks to proper wrappers

Community

- ROS is supported and constantly improved by a large community of engineers and hobbyists from around the globe with a shared interest in robotics and open-source software
- Some useful links:
 - docs.ros.org contains ROS documentation
 - wiki.ros.org webpage provides basic and advanced tutorial to learn how to install and use ROS
 - robotics.stackexchange.com Q&A website allows you to directly ask you solution for your own problems (and contains thousand of questions already answered)
 - discourse.ros.org contains news and general discussion about ROS

ROS philosophy

- **Peer to peer:** individual programs communicate over defined API (ROS messages, services, etc.)
- **Distributed:** programs can be run on multiple computers and communicate over the network
- **Multi-language:** ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.)
- **Light-weight:** stand-alone libraries are wrapped around with a thin ROS layer
- **Free and open-source:** most ROS software is open-source and free to use

ROS master³

- Manages the communication between nodes (processes)
- Every node registers at startup with the master to be able to find each other and exchange messages
- In a distributed system, we should run the master on one computer
- Remote nodes can find each other by communicating with this master

Start a ROS master

```
$ roscore
```

³<http://wiki.ros.org/Master>

ROS nodes

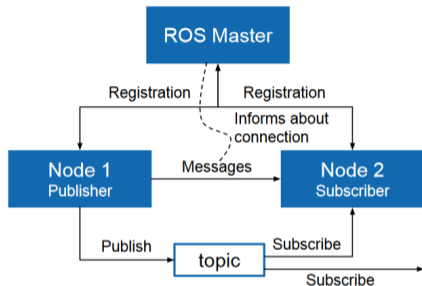
- Nodes are the processes that perform computation (executable)
- ROS nodes are written using ROS client libraries implementing different ROS functionalities such as communication between nodes
- Allow building multiple simple processes rather than a large process with all the functionality (modularity)

Run a ROS node

```
$ rosruntime package_name node_name
```


ROS topics

- Nodes communicate over topics
- Nodes can **publish** or **subscribe** to a topic (1 publisher and n subscribers)
- Topic is a name for a stream of messages



List active topics

```
$ rostopic list
```

Show information about a topic

```
$ rostopic info /topic
```

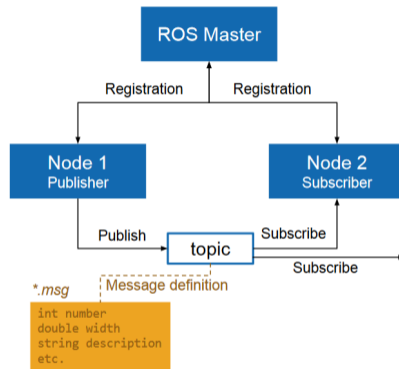
Subscribe and print the contents of a topic

```
$ rostopic echo /topic
```



ROS messages

- Set of standard and custom data structures
- The message definition consists in a typical data structure composed by two main types (fields and constants)
- Defined in *.msg files



Show the type of a topic

```
$ rostopic type /topic
```

Publish a message to a topic

```
$ rostopic pub /topic type data
```

Example ROS message

- `geometry_msgs::PoseStamped` is used to share the timed pose of an object

geometry_msgs/Point.msg

```
float64 x  
float64 y  
float64 z
```

sensor_msgs/Image.msg

```
std_msgs/Header header  
  uint32 seq  
  time stamp  
  string frame_id  
uint32 height  
uint32 width  
string encoding  
uint8 is_bigendian  
uint32 step  
uint8[] data
```

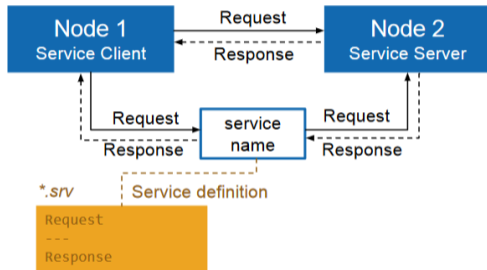
geometry_msgs/PoseStamped.msg

```
std_msgs/Header header  
  uint32 seq  
  time stamp  
  string frame_id  
geometry_msgs/Pose pose  
  geometry_msgs/Point position  
    float64 x  
    float64 y  
    float64 z  
  geometry_msgs/Quaternion orientation  
    float64 x  
    float64 y  
    float64 z  
    float64 w
```



ROS services

- Request/response communication between nodes is realized with services
- The service **server** advertises the service
- The service **client** accesses this service
- Similar in structure to messages, services are defined in *.srv files



List active services

```
$ rosservice list
```

Send a service request

```
$ rosservice call /service data
```

ROS param

- Rosparam allows you to store and manipulate data on the ROS Parameter Server
- The Parameter Server can store integers, floats, boolean, dictionaries, and lists
- Rosparam has many commands that can be used on parameters, as shown below

Set parameter

```
$ rosparam set param_name value
```

Get parameter

```
$ rosparam get param_name
```

List parameter names

```
$ rosparam list
```

Load parameters from file

```
$ rosparam load file_name namespace
```

Publisher/subscriber example

Start a roscore

```
$ roscore
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ roscore
... logging to /home/mrvlvgg/.ros/log/db76e038-65e4-11ee-92ea-73e5bddfaa59/rosla
unch-mrvlvgg-XPS-15-7590-4322.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://mrvlvgg-XPS-15-7590:38451/
ros_comm version 1.16.0

SUMMARY
=====

PARAMETERS
* /roscore: noetic
* /rosversion: 1.16.0

NODES

auto-starting new master
process[master]: started with pid [4357]
ROS_MASTER_URI=http://mrvlvgg-XPS-15-7590:11311/

setting /run_id to db76e038-65e4-11ee-92ea-73e5bddfaa59
process[rosout-1]: started with pid [4378]
started core service [/rosout]
```

Publisher/subscriber example (cont'd)

Run a talker demo node

```
$ rosruncpp roscpp_tutorials talker
```

See the list of active nodes

```
$ rosnodetool list
```

```
mrvlgg@mrvlgg-XPS-15-7590:~$ rosruncpp roscpp_tutorials talker
[ INFO] [1696774625.441541669]: hello world 0
[ INFO] [1696774625.541776791]: hello world 1
[ INFO] [1696774625.641736679]: hello world 2
[ INFO] [1696774625.741712759]: hello world 3
[ INFO] [1696774625.841714213]: hello world 4
[ INFO] [1696774625.941732866]: hello world 5
[ INFO] [1696774626.041707253]: hello world 6
[ INFO] [1696774626.141818579]: hello world 7
[ INFO] [1696774626.241709633]: hello world 8
[ INFO] [1696774626.341705472]: hello world 9
[ INFO] [1696774626.441723468]: hello world 10
[ INFO] [1696774626.541708146]: hello world 11
[ INFO] [1696774626.641698354]: hello world 12
```

```
mrvlgg@mrvlgg-XPS-15-7590:~$ rosnodetool list
/rosout
/talker
```

Publisher/subscriber example (cont'd)

Show information about the *talker* node

```
$ rosnode info /talker
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ rosnode info /talker
-----
Node [/talker]
Publications:
 * /chatter [std_msgs/String]
 * /rosout [roscpp_msgs/Log]

Subscriptions: None

Services:
 * /talker/get_loggers
 * /talker/set_logger_level

contacting node http://mrvlvgg-XPS-15-7590:39255/ ...
Pid: 4687
Connections:
 * topic: /rosout
   * to: /rosout
   * direction: outbound (44645 - 127.0.0.1:32868) [11]
   * transport: TCPROS
```

See information about the *chatter* topic

```
$ rostopic info /chatter
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://mrvlvgg-XPS-15-7590:39255/)

Subscribers: None
```


Publisher/subscriber example (cont'd)

Check the type of the chatter topic

```
$ rostopic type /chatter
```

```
mrs1vgg@mrs1vgg-XPS-15-7590:~$ rostopic type /chatter  
std_msgs/String
```

Show the message contents of the topic

```
$ rostopic echo /chatter
```

```
mrs1vgg@mrs1vgg-XPS-15-7590:~$ rostopic echo /chatter  
data: "hello world 2087"  
---  
data: "hello world 2088"  
---  
data: "hello world 2089"  
---  
data: "hello world 2090"  
---  
data: "hello world 2091"  
---
```

Analyze the frequency

```
$ rostopic hz /chatter
```

```
mrs1vgg@mrs1vgg-XPS-15-7590:~$ rostopic hz /chatter  
subscribed to [/chatter]  
average rate: 10.000  
  min: 0.100s max: 0.100s std dev: 0.00011s window: 10  
average rate: 10.000  
  min: 0.100s max: 0.100s std dev: 0.00015s window: 20  
average rate: 10.000  
  min: 0.100s max: 0.100s std dev: 0.00019s window: 30  
average rate: 10.000  
  min: 0.100s max: 0.100s std dev: 0.00018s window: 40
```

Publisher/subscriber example (cont'd)

Run a listener demo node

```
$ rosrunc roscpp_tutorials listener
```

See the new listener node

```
$ rosnodet list
```

Show the connection of the nodes over the chatter topic

```
$ rostopic info /chatter
```

Close the talker node (Ctrl + C) and publish your own message

```
$ rostopic pub /chatter std_msgs/String "data: 'Robotics Lab Course'"
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ rosrunc roscpp_tutorials listener
[ INFO] [1696775044.341551700]: I heard: [hello world 3294]
[ INFO] [1696775044.441435475]: I heard: [hello world 3295]
[ INFO] [1696775044.541561714]: I heard: [hello world 3296]
[ INFO] [1696775044.641223863]: I heard: [hello world 3297]
[ INFO] [1696775044.740911497]: I heard: [hello world 3298]
[ INFO] [1696775044.841441798]: I heard: [hello world 3299]
[ INFO] [1696775044.941550822]: I heard: [hello world 3300]
[ INFO] [1696775045.041167557]: I heard: [hello world 3301]
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ rosnodet list
/listener
/rosout
/talker
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
 * /talker (http://mrvlvgg-XPS-15-7590:39255/)

Subscribers:
 * /listener (http://mrvlvgg-XPS-15-7590:36861/)
```

```
mrvlvgg@mrvlvgg-XPS-15-7590:~$ rostopic pub /chatter std_msgs/String "data: 'Robotics Lab Course'"
publishing and latching message. Press ctrl-C to terminate
[ INFO] [1696775182.441228168]: I heard: [hello world 4675]
[ INFO] [1696775246.556260537]: I heard: [Robotics Lab Course]
```

ROS 2

- **Communication:** ROS 2 bases its communication on Data Distribution Service
- **Decentralized:** No master node. This, combined with the ROS 2 intra-process API, offers users an enhanced transmission mechanism
- **Multi-platform:** ROS 1 limits itself to Ubuntu or Debian. ROS 2 runs on macOS, Windows, real-time operating system, and other operating systems
- **Real-time control:** ROS 2 makes up for this ROS 1 weakness by improving robots' performance and timeliness of control
- **Multiple robot systems:** ROS 2 improves network performance and support for multi-robot systems
- **Microcontrollers:** ROS 2 supports embedded microcontrollers units such as M7 and ARM-M4

Workspace environment

- The default ROS workspace is `/opt/ros/distro`

To load a workspace

```
$ source /opt/ros/noetic/setup.bash
```

- It is a good practice to add `source` commands to the `.bashrc` file

Workspace environment (cont'd)

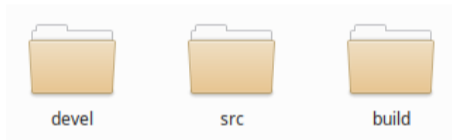
- Working with your workspace

Initialize, build and load a new workspace

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin init
$ cd ..; catkin build
$ source devel/setup.bash
```

- Check your workspace path

```
$ echo $ROS_PACKAGE_PATH
```



src = source code space
 build = cmake and cache info
 devel = targets

Catkin build system

- Catkin is the ROS build system to generate executables, libraries, and interfaces
- It is suggested to use the Catkin Command Line Tools (use `catkin build` instead of `catkin_make` command)

To build a packagee

```
$ catkin build package_name
```

- Whenever you build a new package, update your environment

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

Launch

- launch is a tool for launching multiple nodes (and setting parameters)
- launch files are written in XML and have `.launch` extension
- launch automatically starts a roscore if there is not yet one running

To start a launch file from a folder

```
$ roslaunch file_name.launch
```

To start a launch file from a package

```
$ roslaunch package_name file_name.launch
```

- Example:

```
$ roslaunch roscpp_tutorials talker_listener.launch
```

Launch files

talker_listener.launch

```
<launch>  
  <node name="listener" pkg="roscpp_tutorials" type="listener" output="screen"/>  
  <node name="talker" pkg="roscpp_tutorials" type="talker" output="screen"/>  
</launch>
```

- launch: root element
- node: specifies ad node to be launched
- name: name of the node (free to be chosen)
- pkg: package containing the node
- type: node type (there must be an executable with the same name)
- output: specifies where to output log messages (screen, log)

Launch arguments

- Create re-usable launch files with `<arg>` tag, which works like a parameter

```
<arg name="arg_name" default="default_value"/>
```

- Use arguments in launch file

```
$(arg arg_name)
```

- When launching, arguments can be set

```
$ roslaunch launch_file.launch arg_name:=value
```

range_world.launch (simplified)

```
<launch>

  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find gazebo_plugins)/test/test_worlds/$(arg world).world"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>

</launch>
```

Including other launch files

- Include other launch files with `<include>` tag to organize large projects

```
<include file="package_name"/>
```

- Find the system path to other packages

```
$(find package_name)
```

- Pass arguments to the included file

```
<arg name="arg_name" value="value"/>
```

range_world.launch (simplified)

```
<launch>

  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <include file="$(find gazebo_ros)/launch/
    empty_world.launch">
    <arg name="world_name" value="$(find
      gazebo_plugins)/test/test_worlds/$(arg
      world).world"/>
    <arg name="use_sim_time" value="$(arg
      use_sim_time)"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>

</launch>
```

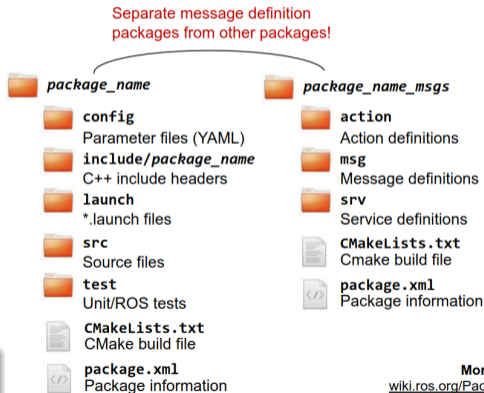


ROS packages

- ROS software is organized into *packages*, which can contain source code, launch files, configuration files, message definitions, data, and documentation
- A package that builds up on/requires other packages (e.g. message definitions), declares these as dependencies

To create a new package

`catkin_create_pkg package_name dependencies`



More info
wiki.ros.org/Packages

ROS packages - package.xml

- The package.xml file defines the properties of the package
 - Package name
 - Version number
 - Authors
 - Dependencies

package.xml

```
<?xml version="1.0"?>
<package format="2">
  <name>ros_package_template</name>
  <version>0.1.0</version>
  <description>A template for ROS packages.</description>
  <maintainer email="name@email.com">Name</maintainer>
  <license>BSD</license>
  <url type="website">https://website</url>
  <author email="name@email.com">Name</author>

  <!-- buildtool_depend: dependencies of the build process -->
  <buildtool_depend>catkin</buildtool_depend>
  <!-- build_depend: dependencies only used in source files -->
  <build_depend>boost</build_depend>
  <!-- depend: build, export, and execution dependency -->
  <depend>roscpp</depend>
</package>
```

ROS packages - CMakeLists.txt

The CMakeLists.txt is input to the CMake build system

1. Required CMake Version (cmake_minimum_required)
2. Package Name (project())
3. Configure C++ standard and compile features
4. Find other CMake/Catkin packages needed for build (find_package())
5. Message/Service/Action Generators (add_message_files(), add_service_files(), add_action_files())
6. Invoke message/service/action generation (generate_messages())
7. Specify package build info export (catkin_package())
8. Libraries/Executables to build (add_library()/add_executable()/target_link_libraries())
9. Tests to build (catkin_add_gtest())
10. Install rules (install())

CMakeLists.txt

```
cmake_minimum_required (VERSION
  3.10.2 )
project(ros_package_template )

# Use C++14, or 11
set(CMAKE_CXX_STANDARD 14 )
set(CMAKE_CXX_STANDARD_REQUIRED
  TRUE )

# Find catkin macros and
  libraries
find_package (catkin REQUIRED
  COMPONENTS
  roscpp
  sensor_msgs
)

...
```

ROS packages - CMakeLists.txt (cont'd)

- Use the same name as in the package.xml
- Use C++11 by default (or 14)
- List the required packages
- Specify build export information
 - INCLUDE_DIRS: Directories with header files
 - LIBRARIES: Libraries created in this project
 - CATKIN_DEPENDS: Packages dependent projects also need
 - DEPENDS: System dependencies dependent projects also need
- Specify locations of header files
- Declare an executable, the node, with two src files
- Specify libraries to link the executable against

CMakeLists.txt example

```

cmake_minimum_required(VERSION 3.10.2)
project(highlevel_controller)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

find_package(catkin REQUIRED
  COMPONENTS roscpp sensor_msgs
)

catkin_package(
  INCLUDE_DIRS include
  # LIBRARIES
  CATKIN_DEPENDS roscpp sensor_msgs
  # DEPENDS
)

include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(${PROJECT_NAME}
src/${PROJECT_NAME}_node.cpp
src/HighlevelController.cpp)

target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})

```

ROS C++ client library (Roscpp)

Essential components of the client library

- Initialization and spinning
- Node handle
- Logging
- Subscriber/publisher
- Parameters
- ...

Roscpp

Initialization and spinning

hello_world.cpp

```
#include <ros/ros.h>

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "hello_world");
    ros::NodeHandle nodeHandle;
    ros::Rate loopRate(10);

    unsigned int count = 0;
    while (ros::ok()) {
        ROS_INFO_STREAM("Hello World " << count);
        ros::spinOnce();
        loopRate.sleep();
        count++;
    }
    return 0;
}
```

- ROS main header file include
- `ros::init(...)` has to be called before other ROS functions
- The node handle is the access point for communications with the ROS system (topics, services, parameters)
- `ros::Rate` is a helper class to run loops at a desired frequency
- `ros::ok()` checks if a node should continue running
- Returns false if SIGINT is received (Ctrl + C) or `ros::shutdown()` has been called
- `ROS_INFO()` logs messages to the filesystem
- `ros::spinOnce()` processes incoming messages via callbacks

Roscpp - Subscriber

listener.cpp

```
#include <ros/ros.h>
#include <std_msgs/String.h>

void chatterCallback(const std_msgs::String& msg)
{
    ROS_INFO("I heard: [%s]", msg.data.c_str());
}

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "listener");
    ros::NodeHandle nodeHandle;

    ros::Subscriber subscriber = nodeHandle.subscribe("
        chatter",10,chatterCallback);
    ros::spin();

    return 0;
}
```

- When a message is received, callback function is called with the contents of the message as argument
- Start listening to a topic by calling the method `subscribe()` of the node handle
- Hold on to the subscriber object until you want to unsubscribe
- `ros::spin()` processes callbacks and will not return until the node has been shutdown

Roscpp - Publisher

talker.cpp

```
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char* argv[]) {
    ros::init(argc, argv, "talker");
    ros::NodeHandle nh;
    ros::Publisher chatterPublisher = nh.advertise<std_msgs::String>("
        chatter", 1);
    ros::Rate loopRate(10);
    unsigned int count = 0;
    while (ros::ok()) {
        std_msgs::String message;
        message.data = "hello world " + std::to_string(count);
        ROS_INFO_STREAM(message.data);
        chatterPublisher.publish(message);
        ros::spinOnce();
        loopRate.sleep();
        count++;
    }
    return 0;
}
```

- Create a publisher with help of the node handle
- Create the message contents
- Publish the contents

Roscpp - Object Oriented Programming

my_package_node.cpp

```
#include <ros/ros.h>
#include "my_package/MyPackage.hpp"

int main(int argc, char* argv[])
{
    ros::init(argc, argv, "my_package");
    ros::NodeHandle nodeHandle("");
    my_package::MyPackage myPackage(nodeHandle);
    ros::spin();
    return 0;
}
```

Class MyPackage



[MyPackage.hpp](#)



[MyPackage.cpp](#)

Main node class providing ROS interface (subscribers, parameters, timers etc.)

Class Algorithm



[Algorithm.hpp](#)



[Algorithm.cpp](#)

Class implementing the algorithmic part of the node - could be separated in a library

NOTE: Specify a function handler to a method from within the class as

```
subscriber_ = nodeHandle_.subscribe(topic, queue_size, &ClassName::methodName, this);
```

ROS Parameter server

- Nodes use the parameter server to store and retrieve parameters at runtime
- Best used for static data such as configuration parameters
- Parameters can be defined in launch files or separate YAML files

List all parameters

```
roscpp param list
```

Get the value of a parameter

```
roscpp param get parameter_name
```

Set the value of a parameter

```
roscpp param set parameter_name value
```

config.yaml

```
camera:  
  left:  
    name: left_camera  
    exposure: 1  
  right:  
    name: right_camera  
    exposure: 1.1
```

package.launch

```
<launch>  
  <node name="name" pkg="package" type="node_type">  
    <roscpp param command="load" file="$(find package)/config/config.yaml" />  
  </node>  
</launch>
```

Roscpp - Parameters

- Get a parameter in C++

List all parameters

```
nodeHandle.getParam(parameter_name, variable)
```

- Method returns true if parameter was found, false otherwise
- Global and relative parameter access

Global parameters

```
nodeHandle.getParam("/package/camera/left/exposure", variable)
```

Relative parameter

```
nodeHandle.getParam("camera/left/exposure", variable)
```

Example code

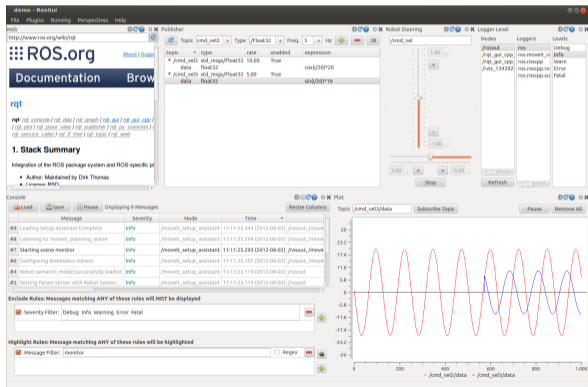
```
ros::NodeHandle nodeHandle("~");  
std::string topic;  
if (!nodeHandle.getParam("topic", topic))  
{  
    ROS_ERROR("Could not find topic  
parameter!");  
}  
ROS_INFO_STREAM("Read topic: " << topic);
```

rqt

- Qt-based software framework for GUI development for ROS
- Various GUI tools in the form of plugins
- One can run all the existing GUI tools as dockable windows within rqt
- Users can create their own plugins for rqt

Run RQT

```
$ rosrunc rqt_gui rqt_gui
```



rqt - plugins

rqt_image_view: provides a GUI plugin for displaying images using `image_transport`

rqt_plot: provides a GUI plugin visualizing numeric values in a 2D plot using different plotting backends

rqt_graph: provides a GUI plugin for visualizing the ROS computation graph

rqt_console: provides a GUI plugin for displaying and filtering ROS messages

rqt_logger_level: provides a GUI plugin for configuring the logger level of ROS nodes

Rviz

- 3D visualization tool for ROS
- Subscribes to topics and visualizes the message contents
- Different camera views (orthographic, top-down, etc.)
- Interactive tools to publish user information
- Save and load setup as RViz configuration
- Extensible with plugins

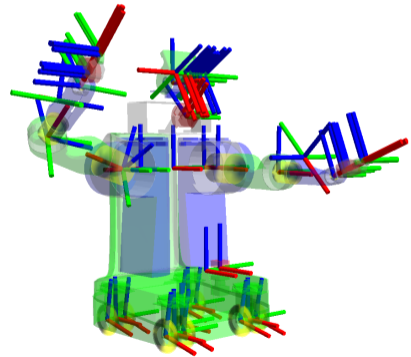


Run Rviz

```
$ rosrun rviz rviz
```


Rviz - Tf

- `tf` is a package that lets the user keep track of multiple coordinate frames over time
- `tf` maintains the relationship between coordinate frames in a tree structure buffered in time
- `tf` lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time



Tf

- The `robot_state_publisher` package allows you to publish the state of a robot
- The package takes the joint angles of the robot as input and publishes the 3D poses of the robot links, using a kinematic tree model of the robot
- It uses the URDF specified by the parameter `robot_description` and the joint positions from the topic `joint_states` to calculate the forward kinematics of the robot and publish the results via `tf`
- Implemented as publisher/subscriber model on the topics `/tf` and `/tf_static`

Tf

- TF listeners use a buffer to listen to all broadcasted transforms
- Query for specific transforms from the transform tree

```
geometry_msgs/TransformStamped[] transforms
std_msgs/Header header
  uint32 seqtime stamp
  string frame_id
string child_frame_id
geometry_msgs/Transform transform
  geometry_msgs/Vector3 translation
  geometry_msgs/Quaternion rotation
```

Tf

Command line

Print info current transform tree

```
$ rosrunc tf tf_monitor
```

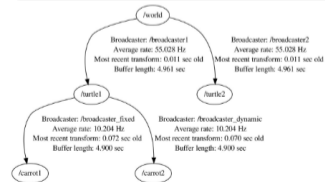
Print info transform between two frames

```
$ rosrunc tf tf_echo
source_frame target_frame
```

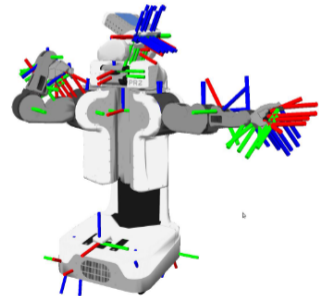
View frames

Creates a visual graph (PDF)

```
$ rosrunc tf2_tools
view_frames.py
```



Rviz



ROS - Time

- Normally, ROS uses the PC's system clock as time source (wall time)
- For simulations or playback of logged data, it is convenient to work with a simulated time (pause, slow-down etc.)
- To work with a simulated clock:
 - Set the `/use_sim_time` parameter to true
 - Publish the time on the topic `/clock` from
 - Gazebo (enabled by default)
 - ROS bag (use option `-clock`)

Set `use_sim_time` parameter

```
$ rosparam set use_sim_time true
```

ROS - Time

- To take advantage of the simulated time, you should always use the ROS Time APIs. `roslib` contains the definition of the `ros::Time` and `ros::Duration` objects used in `roscpp` and other ROS C++ libraries:

`ros::Time`

```
ros::Time begin = ros::Time::now();  
double secs = begin.toSec();
```

`ros::Duration`

```
ros::Duration duration(0.5); // 0.5s  
ros::Duration passed = ros::Time::now() - begin;
```

`ros::Rate`

```
ros::Rate rate(10); // 10Hz
```

ROS - Bags

The rosbag package provides a command-line tool for working with bags as well as code APIs for reading/writing bags in C++ and Python.

- A bag is a format for storing message data
- Binary format with file extension *.bag
- Suited for logging and recording datasets for later visualization and analysis

Record all topics in a bag

```
$ rosbag record --all
```

Record given topics

```
$ rosbag record topic_1 topic_2 topic_3
```

ROS - Bags

- Show information about a bag

```
$ rosbag info bag_name.bag
```

- Read a bag and publish its contents

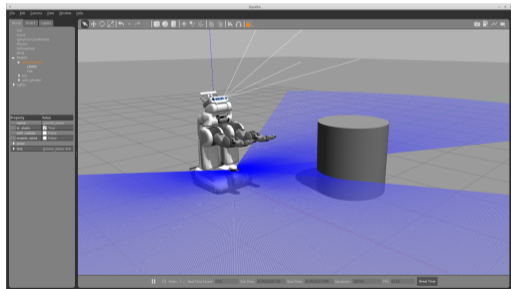
```
$ rosbag play bag_name.bag
```

- Playback options can be defined e.g.

```
$ rosbag play --rate=0.5 bag_name.bag
```


Gazebo

- Simulate 3D rigid-body dynamics
- Simulate a variety of sensors including noise
- 3D visualization and user interaction
- Includes a database of many robots (as Gazebo worlds)
- Provides a ROS interface
- Extensible with plugins

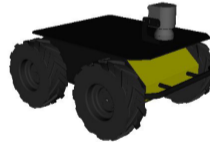


<https://gazebosim.org/>

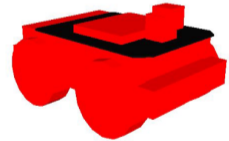
```
$ rosrun gazebo_ros gazebo
```

Unified Robot Description Format - URDF

- Defines an XML format for representing a robot model
 - Kinematic and dynamic description
 - Visual representation
 - Collision model
- URDF generation can be scripted with XACRO macro



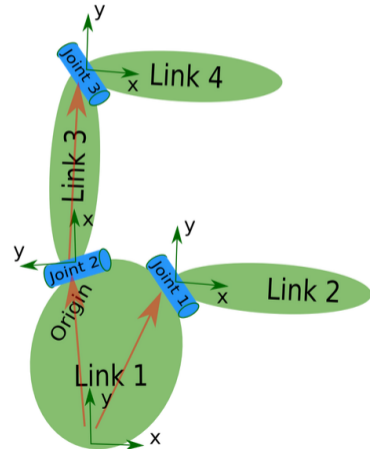
Visual meshes



Primitives for collision

Unified Robot Description Format - URDF (cont'd)

- Description consists of a set of link elements and a set of joint elements
- Joints connect the links together



The <robot> element

```

<robot name="robot_name">
  <!-- robot links and joints and more -->
  <link> ... </link>
  <link> ... </link>

  <joint> .... </joint>
  <joint> .... </joint>
</robot>
  
```

The <joint> element

```

<joint name="my_joint" type="floating">
  <origin xyz="0 0 1" rpy="0 0 3.1416"/>
  <parent link="link1"/>
  <child link="link2"/>

  <calibration rising="0.0"/>
  <dynamics damping="0.0" friction="0.0"/>
  <limit effort="30" velocity="1.0" lower="-2.2" upper="
    0.7" />
  <safety_controller k_velocity="10" k_position="15"
    soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
  
```

The <link> element

```

<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100" ixy="0" izx="0" iyy="100" iyz="0"
      izz="100" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
  
```

Unified Robot Description Format - URDF (cont'd)

- The robot description (URDF) is stored on the parameter server under /robot_description param
- You can visualize the robot model in rviz with the RobotModel plugin

```

<?xml version="1.0"?>
<launch>

  <!-- This launch file just loads the URDF with the given
  hardware interface and robot name into the ROS
  Parameter Server -->
  <arg name="hardware_interface" default="
  PositionJointInterface"/>
  <arg name="robot_name" default="iiwa"/>
  <arg name="origin_xyz" default="'0 0 0'"/> <!-- Note the
  syntax to pass a vector -->
  <arg name="origin_rpy" default="'0 0 0'"/>

  <param name="robot_description" command="$(find xacro)/
  xacro --inorder '$(find iiwa_description)/urdf/iiwa7.
  urdf.xacro' hardware_interface:=$(arg
  hardware_interface) robot_name:=$(arg robot_name)
  origin_xyz:=$(arg origin_xyz) origin_rpy:=$(arg
  origin_rpy)"/>
</launch>

```

Simulation Description Format - SDF

- Defines an XML format to describe
 - Environments (lighting, gravity etc.)
 - Objects (static and dynamic)
 - Sensors
 - Robots
- SDF is the standard format for Gazebo
- Gazebo automatically converts a URDF to SDF



Xacro

- Xacro is an XML macro language
- Used to construct shorter and more readable XML files by using macros
- It is heavily used in packages such as the urdf

Example Xacro

```
<xacro:macro name="pr2_arm" params="suffix parent reflect">
  <pr2_upperarm suffix="${suffix}" reflect="${reflect}"
    parent="${parent}" />
  <pr2_forearm suffix="${suffix}" reflect="${reflect}"
    parent="elbow_flex_${suffix}" />
</xacro:macro>

<xacro:pr2_arm suffix="left" reflect="1" parent="torso" />
<xacro:pr2_arm suffix="right" reflect="-1" parent="torso"
  />
```

Xacro (cont'd)

- Properties are named values or named blocks that can be inserted anywhere into the XML document
- Properties can be manually declared or loaded from YAML files

- Macros may contain other macros
- You can include other xacro files using the `xacro:include` tag

Xacro properties

```
<xacro:property name="front_left_origin">  
  <origin xyz="0.3 0 0" rpy="0 0 0" />  
</xacro:property>  
  
<pr2_wheel name="front_left_wheel">  
  <xacro:insert_block name="front_left_origin" />  
</pr2_wheel>
```

Xacro include

```
<xacro:include filename="$(find package)/other_file.xacro"  
  />  
<xacro:include filename="other_file.xacro" />  
<xacro:include filename="$(cwd)/other_file.xacro" />
```


Xacro (cont'd)

- Convert xacro to urdf from command line

```
$ rosrun xacro xacro robot_name.xacro > robot_name.urdf
```

- ... or inside a launch file

```
<param name="robot_description" command=" $(find xacro)/xacro $ (find  
robot_description_pkg)/urdf/robot_name.xacro" />
```

Gazebo

- Robot simulator integrated in ROS via
 - `gazebo_ros_pkgs`: wrappers, tools and additional API's for using ROS with the Gazebo simulator
 - `gazebo-msgs`: Message and service data structures for interacting with Gazebo from ROS
 - `gazebo-plugins`: provide functionality for the simulation of sensors and actuators
 - `gazebo-ros-control`: simulated controllers to actuate the joints of your robot

Check gazebo installation

```
$ rosrun gazebo_ros gazebo
```

Retrieve model information

```
$ rostopic echo /gazebo/model_states
```

Kuka iiwa example

```
$ roslaunch iiwa_gazebo iiwa_gazebo.  
launch trajectory:=false
```

Gazebo plugins

Give your URDF models greater functionality

Shared libraries that have direct access to all the functionalities of Gazebo

There are currently 6 types of plugins:

- **World:** to control world properties
- **Model:** to control the joints and the state
- **Sensor:** to acquire sensor information and control sensor properties
- **System:** specified at the command line to give the user control over the startup process
- **Visual:** to access the visual rendering functions
- **GUI:** loaded on startup to control the GUI

ROS - Sensors

ROS sensor plugins have been implemented to wrap the Gazebo sensor plugins

The available ROS sensor plugins are available in the `gazebo_plugins` of `gazebo_ros_pkgs`, like those related to

- Camera
- Depth
- Lidar
- IMU

The camera ROS plugin

- The camera ROS plugin provides the ROS interface for simulating cameras
- It publishes the `sensor_msgs::CameraInfo` and `sensor_msgs::Image` ROS messages
- The ROS camera plugin is `libgazebo_ros_camera.so`

camera plugin

```
<plugin name="camera_controller" filename="
  libgazebo_ros_camera.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>0.0</updateRate>
  <cameraName>camera</cameraName>
  <imageTopicName>image_raw</imageTopicName>
  <cameraInfoTopicName>camera_info</cameraInfoTopicName>
  <frameName>camera_link_optical</frameName>
  <hackBaseline>0.0</hackBaseline>
  <distortionK1>0.0</distortionK1>
  <distortionK2>0.0</distortionK2>
  <distortionK3>0.0</distortionK3>
  <distortionT1>0.0</distortionT1>
  <distortionT2>0.0</distortionT2>
  <CxPrime>0</CxPrime>
  <Cx>0.0</Cx>
  <Cy>0.0</Cy>
  <focalLength>0.0</focalLength>
</plugin>
```

The camera ROS plugin

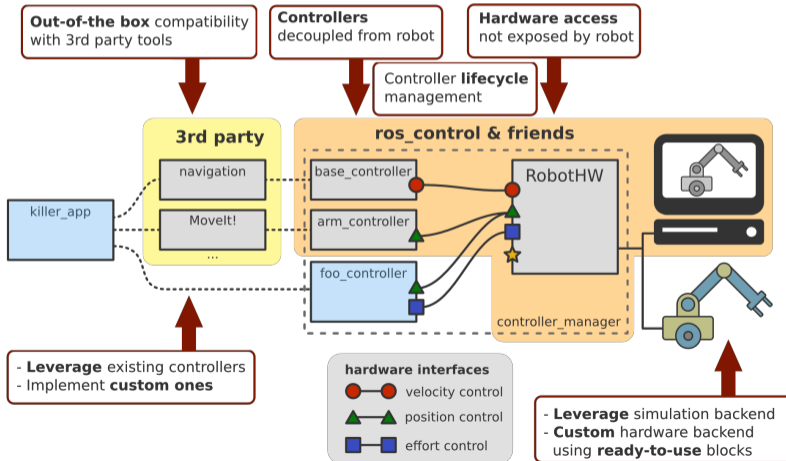
- To enable a camera, first add a camera_link to your urdf
- Then, include the <sensor> element and the load the plugin within the associated <gazebo> tag

camera gazebo

```

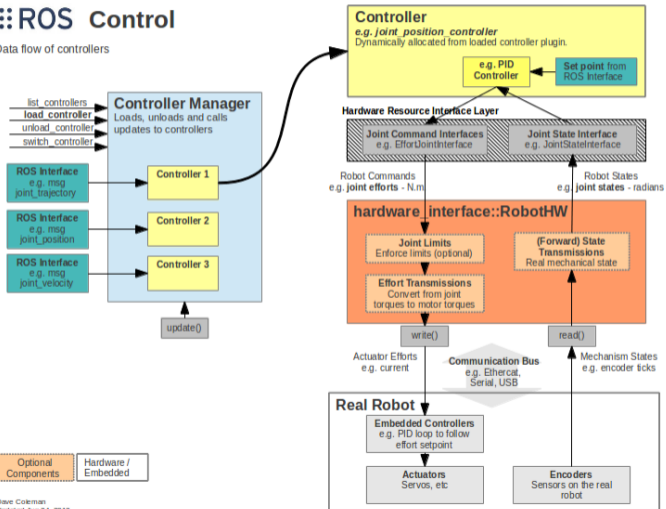
<gazebo reference="camera_link">
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width> <height>800</height> <format>
          R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near> <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type> <mean>0.0</mean> <stddev>0.007
        </stddev>
      </noise>
    </camera>
    <plugin name="camera_controller" filename="
      libgazebo_ros_camera.so"> ... </plugin>
  </sensor>
</gazebo>
  
```

The `ros_control` framework provides the capability to implement and manage robot controllers



ROS Control

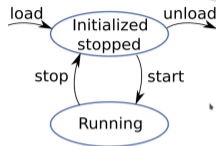
Data flow of controllers



Available controllers

- `joint_state_controller` defined to publish joint states
- `joint_position_controller` position commands are used to control joint positions
- `joint_velocity_controller` velocity commands are used to control joint positions or velocities
- `joint_effort_controller` efforts commands are used to control joint positions, velocities or efforts
- `joint_trajectory_controllers` used to control the execution of joint-space trajectories on a group of joints

Controller management The `controller_manager` provides the infrastructure to interact with controllers, i.e. load, unload, start and stop controllers



Command line

```
$ rosrun controller_manager controller_manager <command> <controller_name> or
$ rosrun controller_manager spawner [--stopped] name1 name2 name3
```

Launch file

```
<node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="
false" output="screen" args="$(arg controllers)" />
```

Using rqt

```
$ rosrun rqt_controller_manager rqt_controller_manager
```

Configuring and launching controllers

Controllers are usually defined with yaml files

The example shows

- the joint state controller to publish the joint states of the arm (with a publishing rate set to 50 Hz)
- two joint position controllers to move each of the two joints upon receiving a goal position

rrbot_control.yaml

```
rrbot:
  # Publish all joint states -----
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Position Controllers -----
  joint1_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint1
    pid: {p: 100.0, i: 0.01, d: 10.0}
  joint2_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint2
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

Configuring and launching controllers

- The yaml configuration file can be loaded in a launch file, prior to the spawning of the controller
- To simulate the robot in Gazebo, the corresponding plugin must be added to the URDF model

rrbot_control.launch

```
<launch>
  <!-- Load joint controller configurations from YAML file to parameter
  server -->
  <rosparam file="$(find rrbot_control)/config/rrbot_control.yaml"
  command="load"/>

  <!-- load the controllers -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner"
  respawn="false"
  output="screen" ns="/rrbot" args="joint_state_controller
  joint1_position_controller joint2_position_controller"/>

  <!-- convert joint states to TF transforms for rviz, etc -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
  robot_state_publisher"
  respawn="false" output="screen">
    <remap from="/joint_states" to="/rrbot/joint_states" />
  </node>
</launch>
```

Hardware abstraction layer

ros_control interfaces

The connection of the controllers with the hardware interface (that encapsulate the hardware) is done through the ros_control interfaces

Common interfaces `ros_control` interfaces are

- Joint State Interfaces: to retrieve the joint states from the actuators encoder.
- Joint Command Interfaces
 - `EffortJointInterface`: to send the effort command
 - `VelocityJointInterface`: to send the velocity command
 - `PositionJointInterface`: to send the position command

Hardware abstraction layer

Joint Limits

- The `joint_limits_interface` package contains data structures for representing joint limits, methods to populate them through URDF or yaml files, and methods to enforce these limits
- Joint limits can be specified in URDF and in YAML format

joint limits URDF

```
<joint name="{robot_name}_joint_1" type="revolute">
  ...
  <!-- Joint limits -->
  <limit lower="{-120 * PI / 180}" upper="{120 * PI / 180}"
        effort="{max_effort}" velocity="{max_velocity}"
  />
  <!-- Soft limits -->
  <safety_controller soft_lower_limit="{-118 * PI / 180}"
                    soft_upper_limit="{118 * PI / 180}" k_position="{
                    safety_controller_k_pos}" k_velocity="{
                    safety_controller_k_vel}"/>
</joint>
```

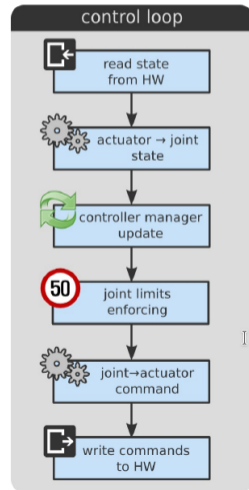
Hardware abstraction layer

Transmissions

interfaces to implement mechanical transmissions, like a mechanical reducer with a given ratio n , thus mapping input/output effort/flow variables while preserving power

`transmission.xacro`

```
<transmission name="{robot_name}_tran_1">
  <robotNamespace>/{robot_name}</robotNamespace>
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="{robot_name}_joint_1">
    <hardwareInterface>hardware_interface/{hardware_interface}</hardwareInterface>
  </joint>
  <actuator name="{robot_name}_motor_1">
    <hardwareInterface>hardware_interface/{hardware_interface}</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```



Gazebo_ros_controllers

To use ros_control within Gazebo, the URDF model of the robot has to include two additional elements: transmissions and the plugin

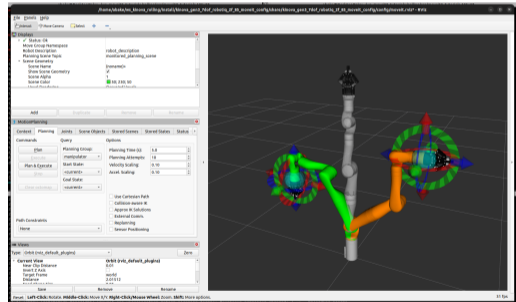
gazebo_ros_control plugin

```
<!-- Load Gazebo lib and set the robot namespace -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/${robot_name}</robotNamespace>
  </plugin>
</gazebo>
```


MoveIt

Most widely used software for manipulation

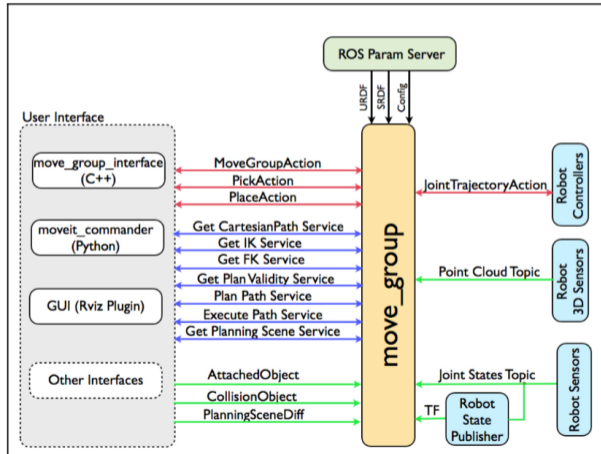
- Motion planning
- Manipulation
- Inverse kinematics
- Control
- 3D perception
- Collision checking



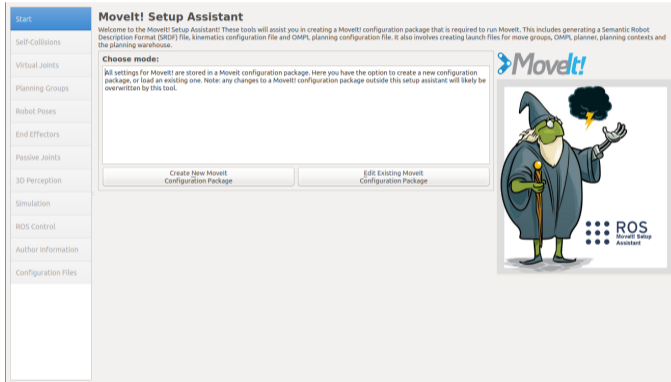
Launch the iiwa moveit

`roslaunch iiwa_moveit demo.launch`

Movelt



How to create the MoveIt config package



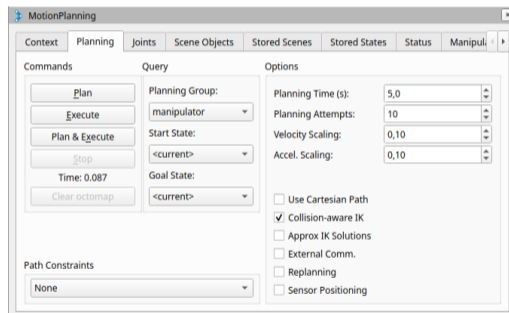
The screenshot shows the MoveIt! Setup Assistant interface. On the left is a vertical sidebar with a blue 'Start' button at the top and several menu items: Self-Collisions, Virtual Joints, Planning Groups, Robot Poses, End Effectors, Passive Joints, 3D Perception, Simulation, ROS Control, Author Information, and Configuration Files. The main area is titled 'MoveIt! Setup Assistant' and contains a welcome message: 'Welcome to the MoveIt! Setup Assistant! These tools will assist you in creating a MoveIt! configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPF planning configuration file. It also involves creating launch files for move groups, OMPF, planner, planning contexts and the planning warehouse.' Below this is a 'Choose mode:' section with a text box containing instructions: 'All settings for MoveIt! are stored in a MoveIt! configuration package. Here you have the option to create a new configuration package, or load an existing one. Note: any changes to a MoveIt! configuration package outside this setup assistant will likely be overwritten by this tool.' At the bottom of this section are two buttons: 'Create New MoveIt Configuration Package' and 'Edit Existing MoveIt Configuration Package'. To the right of the text box is the MoveIt! logo and a cartoon wizard character holding a staff and a lightning bolt, with the text 'ROS MoveIt! Setup Assistant' below it.

Launch Setup Assistant tool

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

The MoveIt planning scene

- Query start state (orange)
- Query goal state (green)
- Interactive markers
- Joints tab - nullspace
- Motion planning
- Plan visualization



Activate *Use Cartesian Path* checkbox

The robot will attempt to move the end effector linearly in cartesian space

Motion control of robotic manipulators

Kinematics and dynamics library - KDL

Robotic vision

Introduction

- Computer Vision is an essential part of robotics
- It helps the robot extract information from camera data to understand its environment
- Applications include
 - extracting an object and its position
 - inspecting manufactured parts for production errors
 - detecting pedestrians in autonomous driving applications
 - make a robot arm perform a somewhat intelligent pick and place task



Interface the sensor

- To program camera sensors we need to interface them to the onboard computer of the robot
- This can be made mainly in two ways
 - Using operating system driver
 - Vendor driver
- Standard USB camera (like webcams) are directly accessible using low level routine provided by the operating system
- In Ubuntu/Linux, after plugging in the camera, check whether a `/dev/videoX` device file has been created

Check camera device file

```
$ ls /dev/ | grep video
```

The `usb_cam` package

- We can install a ROS webcam driver called `usb_cam` using the following command

```
$ sudo apt-get install ros-noetic-usb-cam
```

- Launch this node using the launch file provided with the package

```
$ roslaunch usb_cam usb_cam-test.launch
```

- Show the image using `image_view` package

```
$ rqt_image_view
```

The usb_cam package

```
<launch>
  <!-- start video device with settings -->
  <node name="usb_cam" pkg="usb_cam" type="usb_cam_node"
    output="screen" >
    <param name="video_device" value="/dev/video0" />
    <param name="image_width" value="640" />
    <param name="image_height" value="480" />
    <param name="pixel_format" value="yuyv" />
    <param name="color_format" value="yuv422p" />
    <param name="camera_frame_id" value="usb_cam" />
    <param name="io_method" value="mmap"/>
  </node>
  <!-- display the content of the camera images -->
  <node name="image_view" pkg="image_view" type="image_view"
    respawn="false" output="screen">
    <remap from="image" to="/usb_cam/image_raw"/>
    <param name="autosize" value="true" />
  </node>
</launch>
```

Generated topics

```
/usb_cam/camera_info
/usb_cam/image_raw
/usb_cam/image_raw/compressed
/usb_cam/image_raw/compressed/parameter_descriptions
/usb_cam/image_raw/compressed/parameter_updates
/usb_cam/image_raw/compressedDepth
/usb_cam/image_raw/compressedDepth/parameter_descriptio
/usb_cam/image_raw/compressedDepth/parameter_updates
/usb_cam/image_raw/theora
/usb_cam/image_raw/theora/parameter_descriptions
/usb_cam/image_raw/theora/parameter_updates
```

The image_transport package

- The compressed format is useful to send images to other ROS nodes over the network or store video data into bagfiles
- These topics are published by the image_transport package that provides transparent support for transporting images in low-bandwidth compressed formats
- Its internal mechanism is very similar to using ROS Publishers and Subscribers

```
// Use the image_transport classes instead.
#include <ros/ros.h>
#include <image_transport/image_transport.h>

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{ ... }

ros::NodeHandle nh;
image_transport::ImageTransport it(nh);
image_transport::Subscriber sub = it.subscribe("in_image_base_topic", 1, imageCallback);
image_transport::Publisher pub = it.advertise("out_image_base_topic", 1);
```

The image_transport package

- To use the compressed image we need to republish it in an uncompressed format, using the republish node of the image_transport package
- Suppose we have recorded the compressed video stream

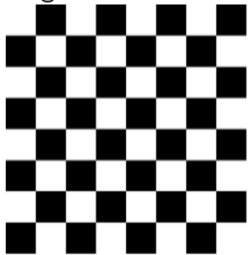
```
rosviz record /usb_cam/image_raw/compressed
```

- To republish it uncompressed

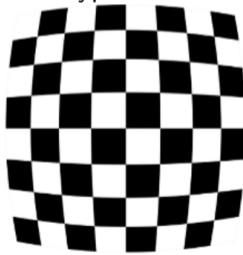
```
$ rosviz image_transport republish compressed in:=usb_cam/image_raw/compressed raw  
out:=usb_cam/image_raw
```

Camera calibration

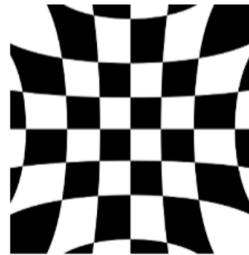
- Cameras need calibration to correct the distortions in the camera images due to the camera's internal parameters
- The next figure shows two common types of radial distortion



No distortion



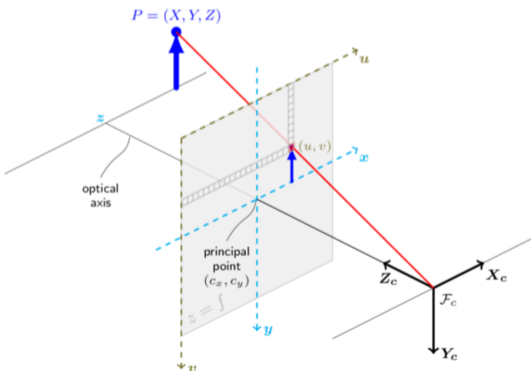
Positive radial distortion
(Barrel distortion)



Negative radial distortion
(Pincushion distortion)

Camera calibration

- We assume to have a standard pinhole camera



$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

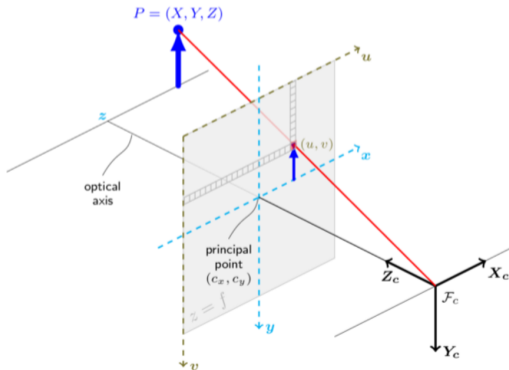
$$y' = y/z$$

$$u = f_x * x' + c_x$$

$$v = f_y * y' + c_y$$

Camera calibration

- We assume to have a standard pinhole camera



$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$x'' = x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2 (r^2 + 2x'^2)$$

$$y'' = y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1 (r^2 + 2y'^2) + 2p_2 x' y'$$

$$\text{where } r^2 = x'^2 + y'^2$$

$$u = f_x * x'' + c_x$$

$$v = f_y * y'' + c_y$$

Camera calibration

- The `camera_calibration` package allows easy calibration of monocular or stereo cameras using a checkerboard calibration target
- To run the `cameracalibrator.py` node for a monocular camera using an 8x6 chessboard with 108mm squares

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 image:=/  
usb_cam/image_raw /image camera:=/usb_cam
```

Camera calibration

- `image_proc` package removes camera distortion from the raw image stream



- It is meant to sit between the camera driver and vision processing nodes

To run `image_proc`

```
$ ROS_NAMESPACE=usb_cam rosrn image_proc image_proc
```

OpenCV - Open Computer Vision Library

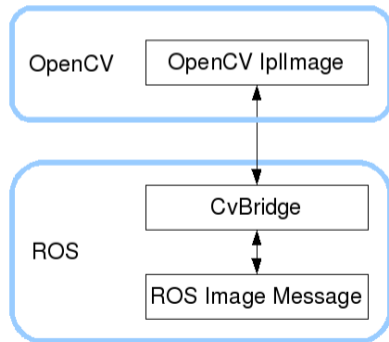
- OpenCV - open-source library that provides real-time optimized Computer Vision algorithms, tools, and hardware. It also supports model execution for Machine Learning (ML)
- `vision_opencv` - stack for interfacing ROS with OpenCV via `cv_bridge` package
- Using OpenCV in your ROS code

In your `CMakeLists.txt`

```
find_package(OpenCV)
include_directories (${OpenCV_INCLUDE_DIRS})
target_link_libraries(my_awesome_library ${OpenCV_LIBRARIES})
```

OpenCV - CvBridge

- The CvBridge library acts as a bridge for converting the ROS messages to OpenCV images and vice versa
- ROS passes around images in its own `sensor_msgs/Image` message format
- Use CvBridge to convert ROS images into OpenCV `cv::Mat` format



OpenCV - CvBridge example code

- Create a new package with

```
$ catkin_create_pkg opencv_example cv_bridge image_transport roscpp roslib
```

- In your CMakeLists.txt uncomment

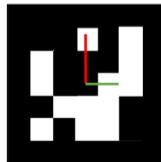
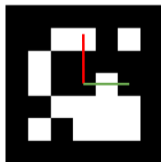
```
add_executable($ {PROJECT_NAME}_node src/opencv_example_node.cpp)
...
target_link_libraries($ {PROJECT_NAME}_node ...
```

- Create and put your code inside the file `src/opencv_example_node.cpp`^a

^ahttp://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages

Fiducial Markers

- Efficient algorithms to perform object recognition and pose estimation working in real world environments are difficult to implement
- In many cases one camera is not enough to retrieve the three-dimensional pose of an object
- Markers are typically represented by a synthetic square image composed by a wide black border and an inner binary matrix which determines its unique identifier



Fiducial Markers

- When the *intrinsic parameters* of the camera and the size of the fiducial are known, the pose of the fiducial relative to the camera can be estimated
- The pose estimation code solves a set of linear equations to determine the world (X, Y, Z) coordinate of each of the vertices
- From this, we obtain the *transform* of the fiducial's coordinate system to the camera's coordinate system
- A robot can determine its position and orientation by looking at a number of fiducial markers

Fiducial Markers

- To install the fiducial software from binary packages

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

^a<http://wiki.ros.org/fiducials>

- Two nodes should be run, `aruco_detect`, which handles the detection of the fiducials, and `fiducial_slam`, which combines the fiducial pose estimates and builds the map and makes an estimate of the robot's position.

```
$ roslaunch aruco_detect aruco_detect.launch
```

```
$ roslaunch fiducial_slam fiducial_slam.launch
```


Fiducial Markers - aruco_ros

- To use the aruco_ros fiducial packages, clone it from the repo⁴
- Checkout the correct branch

```
$ cd catkin_ws/src/aruco_ros  
$ git checkout noetic-devel
```

- Compile and run to check if it works correctly

```
$ catkin build  
$ roslaunch aruco_ros single.launch
```

⁴https://github.com/pal-robotics/aruco_ros.git

Fiducial Markers - aruco_ros

- You should then subscribe to your camera topic
- Create a launch file as the one shown here

```

<launch>

  <arg name="markerId"          default="201"/>
  <arg name="markerSize"        default="0.1"/> <!-- in m -->
  <arg name="camera"            default="usb_cam"/>
  <arg name="marker_frame"      default="aruco_marker_frame"/>
  <arg name="ref_frame"         default=""/>
  <arg name="corner_refinement" default="LINES" />

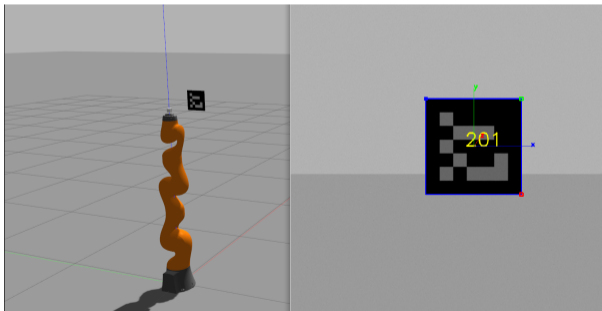
  <node pkg="aruco_ros" type="single" name="aruco_single">
    <remap from="/camera_info" to="/${arg camera}/camera_info" />
    <remap from="/image" to="/${arg camera}/image_raw" />
    <param name="image_is_rectified" value="True"/>
    <param name="marker_size"        value="${arg markerSize}"/>
    <param name="marker_id"          value="${arg markerId}"/>
    <param name="reference_frame"    value="${arg ref_frame}"/>
    <param name="camera_frame"       value="stereo_gazebo_${arg camera}
    _camera_optical_frame"/>
    <param name="marker_frame"       value="${arg marker_frame}" />
    <param name="corner_refinement"  value="${arg corner_refinement}" />
  </node>

</launch>

```

Fiducial Markers

- We can also test and use marker detectors in simulations using Gazebo ROS



Fiducial Markers

- To use marker detectors in simulations using Gazebo ROS
- Generate Aruco marker <https://chev.me/arucogen/>
- Create a Gazebo model, add it to the Gazebo model path

```
<env name="GAZEBO_MODEL_PATH" value="$ (find package)/models:$ (optenv  
GAZEBO_MODEL_PATH) "/>
```

- Start the simulation, import the aruco model into your world, and save the world with name
- Relaunch the simulation loading the new world

```
<include file="$ (find gazebo_ros)/launch/empty_world.launch">  
<arg name="world_name" value="$ (find package)/worlds/aruco.world"/>
```

Point clouds

- A point cloud is a set of data points in space Point clouds are generally produced by 3D scanners or depth sensors, which measure many points on the external surfaces of objects around them
- As the output of 3D scanning processes, point clouds are used for many purposes, including to create 3D CAD models for manufactured parts and quality inspection, and for a multitude of visualization, animation, rendering and mass customization applications

Point clouds

- Point cloud data can be defined as a group of data points in some coordinate system. In 3D, it has x , y , and z coordinates
- One of the most famous framework used to access and elaborate clouds is the Point Cloud Library (PCL) library
- This is an open-source project for handling 2D/3D images and point cloud processing. It consists of standard algorithms for filtering, segmentation, feature estimation, and so on
- PCL is integrated into ROS for handling point cloud data from various sensors

Point clouds - useful packages

- `pcl_conversions`: package providing APIs to convert PCL data types to ROS messages and vice versa
- `pcl_msgs`: contains the definition of PCL-related messages in ROS
- `pcl_ros`: PCL bridge of ROS. This package contains tools and nodes to bridge ROS messages to PCL data types and vice versa
- `pointcloud_to_laserscan`: useful to convert the 3D point cloud into a 2D laser scan, used for ground mobile robot navigation

Point clouds - devices

- Most used sensors in robotics are
 - Microsoft kinect
 - Intel realsense
- Both such device require to install proper libraries to be interfaced with your (or robot) computer

OpenNI (Open source Natural Interaction) - microsoft kinect

```
$ sudo apt-get install ros-noetic-openni2-camera ros-noetic-openni2-launch
```

Real sense ROS wrapper for intel real sense

```
$ sudo apt-get install ros-noetic-realsense2-camera
```


Point clouds - messages

- The current data structures in ROS that represent point clouds is `sensor_msgs::PointCloud`
- Contains several fields
 - `x` - the X Cartesian coordinate of a point (float32)
 - `y` - the Y Cartesian coordinate of a point (float32)
 - `z` - the Z Cartesian coordinate of a point (float32)
 - `rgb` - the RGB (24-bit packed) color at a point (uint32)
 - `rgba` - the A-RGB (32-bit packed) color at a point (uint32)
 - `normal_x` - the first component of the normal direction vector at a point (float32)
 - `normal_y` - the second component of the normal direction vector at a point (float32)
 - `normal_z` - the third component of the normal direction vector at a point (float32)
 - `curvature` - the surface curvature change estimate at a point (float32)
 - ...

Point clouds - Publishing and subscribing to point cloud messages

■ Publisher node

```
// assume you get a point cloud message somewhere
sensor_msgs::PointCloud cloud_msg;

// advertise
ros::Publisher pub = nh.advertise<sensor_msgs::PointCloud>(topic, queue_size);
// and publish
pub.publish(cloud_msg);
```

■ Subscriber node

```
// callback signature:
void callback(const sensor_msgs::PointCloudConstPtr&);

// create subscriber:
ros::Subscriber sub = nh.subscribe(topic, queue_size, callback);
```

Point clouds - real sense

```

<launch>
  <arg name="serial_no"           default=""/>
  <arg name="usb_port_id"        default=""/>
  <arg name="device_type"        default=""/>
  <arg name="json_file_path"     default=""/>
  <arg name="camera"             default="camera"/>
  <arg name="tf_prefix"          default="$(arg camera)"/>
  ...
  <arg name="gyro_fps"           default="-1"/>
  <arg name="accel_fps"          default="-1"/>
  <arg name="enable_gyro"        default="false"/>
  <arg name="enable_accel"       default="false"/>
  ...
  <arg name="enable_pointcloud"  default="false"/>
  <arg name="publish_tf"         default="true"/>
  <arg name="tf_publish_rate"    default="0"/>
  ...
  <group ns="$(arg camera)">
    <include file="$(find realsense2_camera)/launch/
      includes/nodelet.launch.xml">
      ...
    </include>
  </group>
</launch>

```

Generated topics

```

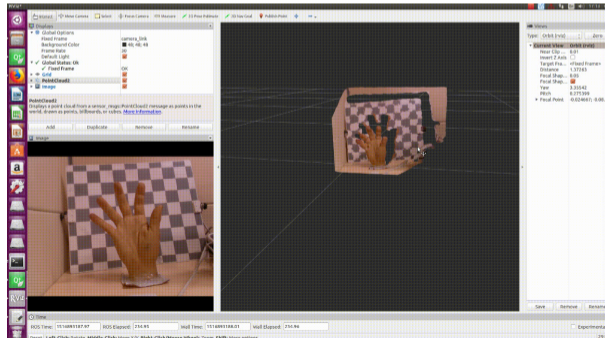
/camera/color/camera_info
/camera/color/image_raw
/camera/color/metadata
/camera/depth/camera_info
/camera/depth/image_rect_raw
/camera/depth/metadata
/camera/extrinsics/depth_to_color
/camera/extrinsics/depth_to_infra1
/camera/extrinsics/depth_to_infra2
/camera/infra1/camera_info
/camera/infra1/image_rect_raw
/camera/infra2/camera_info
/camera/infra2/image_rect_raw
/camera/gyro/imu_info
/camera/gyro/metadata
/camera/gyro/sample
/camera/accel/imu_info
/camera/accel/metadata
/camera/accel/sample
/diagnostics

```

Point clouds - real sense

- Start the camera node and make it publish the point cloud using

```
$ roslaunch realsense2_camera rs_camera.launch filters:=pointcloud
```
- Open rviz to watch the pointcloud



Visual Servoing

- The objective of visual servoing is to ensure that the end-effector, on the basis of visual measurements elaborated in real time, reaches and keeps a (constant or time-varying) desired pose with respect to the observed object

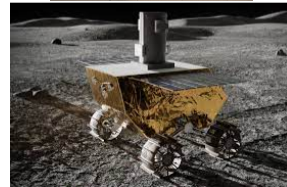
Visual Servoing

- Depending in which space measurements are taken
 - Position-based visual servoing:
 - information extracted from images (features) is used to reconstruct the current 3D pose (pose/orientation) of an object
 - combined with the knowledge of a desired 3D pose, we generate a Cartesian pose error signal that drives the robot to the goal
 - Image-based visual servoing
 - error is computed directly on the values of the features extracted on the 2D image plane, without going through a 3D reconstruction
 - the robot should move so as to bring the current image features (what it “sees” with the camera) to their desired values

Control of mobile robots

Introduction

- Mobile robots have become very important in advanced applications thanks to their potential for autonomous intervention
- Nowadays a lot of mobile robots are commercial and able to work in collaboration with humans
- Applications include
 - autonomous exploration
 - search and rescue operations
 - human-robot collaboration



Tassonomy of mobile robots

- Wheeled ground robots



- Underwater robots



- Aerial robots

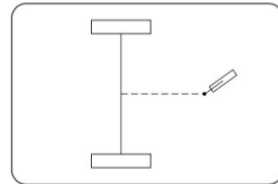
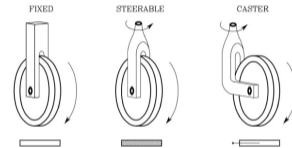


- Legged robots



Wheeled ground robots

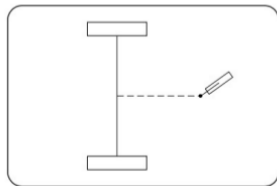
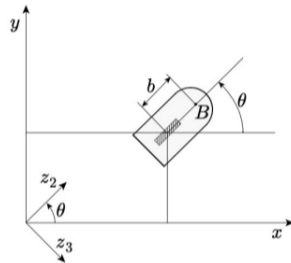
- There are different kind of configurations, in view of the type and number of wheels
 - fixed wheel
 - steerable wheel
 - caster wheel
- One of the most common configuration is the differential drive robot
 - one caster wheel + two fixed actuated wheels
 - two independent motor in the fixed wheels
 - rotation axis in common



Differential drive robot

- Starting from the *unicycle*
- The inputs of this system are:
 - v the heading velocity
 - ω the steering velocity
- A balance problem is evident: differential drive robot is kinematically equivalent
- (x,y) of the unicycle corresponds to the midpoint of the segment that joins the two fixed wheels of the differential drive
- ω_L and ω_R are the left and right wheel velocity, ρ_w is the wheel radius

$$v = \frac{\omega_R + \omega_L}{2} \rho_w \quad \omega = \frac{\omega_R - \omega_L}{d} \rho_w$$



Differential drive robot - Gazebo plugin

- The behavior of the differential drive robot is obtained in simulation using the differential_drive_controller
- Its main task is to take as input a velocity command and it computes the wheels' velocities using differential formulas
- Additionally it publishes an odometry topic that contains values computed from the feedback of the wheel encoder

differential_drive_controller plugin

```

<gazebo>
  <plugin name="differential_drive_controller" filename="
    libgazebo_ros_diff_drive.so">
    <legacyMode>true</legacyMode>
    <rosDebugLevel>Debug</rosDebugLevel>
    <publishWheelTF>false</publishWheelTF>
    <robotNamespace>/</robotNamespace>
    <publishTf>1</publishTf>
    <publishWheelJointState>true</publishWheelJointState>
    <alwaysOn>true</alwaysOn>
    <updateRate>100.0</updateRate>
    <leftJoint>left_wheel_joint</leftJoint>
    <rightJoint>right_wheel_joint</rightJoint>
    <wheelSeparation>${wheel_axle}</wheelSeparation>
    <wheelDiameter>${2*wheel_radius}</wheelDiameter>
    <broadcastTF>0</broadcastTF>
    <wheelTorque>30</wheelTorque>
    <wheelAcceleration>1.8</wheelAcceleration>
    <commandTopic>cmd_vel</commandTopic>
    <odometryFrame>odom</odometryFrame>
    <odometryTopic>odom</odometryTopic>
    <robotBaseFrame>base_footprint</robotBaseFrame>
  </plugin>
</gazebo>
  
```

Velocity command

- The input of a wheeled ground robot is a velocity command
- It contains two geometry_msgs/Vector3, one for the linear velocities and the other for the angular velocities
- Which are the fields that we can fill with values of this message for the differential drive robot?

```
$ rosmg info geometry_msgs/  
Twist
```

```
geometry_msgs/Vector3 linear  
float64 x  
float64 y  
float64 z  
geometry_msgs/Vector3  
angular float64 x  
float64 y  
float64 z
```

Odometry message

- The message has three components:

- A header message where the time stamp the frame_id is showed while the child_frame_id is not part of the header
- A geometry_msgs/PoseWithCovariance that presents a position, and orientation represented as a unit quaternion and a vector of 36 covariances elements
- A geometry_msgs/TwistWithCovariance that presents a linear velocity, angular velocity, and a vector of 36 covariances elements

```
$ rosmmsg info nav_msgs/  
Odometry
```

```
std_msgs/Header header  
uint32 seq  
time stamp  
string frame_id  
string child_frame_id
```

Odometry message

```
geometry_msgs/PoseWithCovariance
  pose
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion
        orientation
          float64 x
          float64 y
          float64 z
          float64 w
        float64[36] covariance
```

```
geometry_msgs/TwistWithCovariance
  twist
    geometry_msgs/Twist twist
      geometry_msgs/Vector3 linear
        float64 x
        float64 y
        float64 z
      geometry_msgs/Vector3 angular
        float64 x
        float64 y
        float64 z
      float64[36] covariance
```

Laser range finder - Gazebo plugin

- The most used sensors for mobile robots are lidars
- There are different types: 2D, 3D, 360° TOF, ecc...
- It produces a sensor_msgs/LaserScan that contains distances from the obstacles



```
<plugin name="gazebo_ros_head_hokuyo_controller" filename="
  libgazebo_ros_laser.so">
  <topicName>/laser/scan</topicName>
  <frameName>laser_frame</frameName>
</plugin>
```


Laser range finder - sensor_msgs/LaserScan

- The laser scan message is composed by

- an header
- a set of parameters
- vector of distances
- vector of intensities

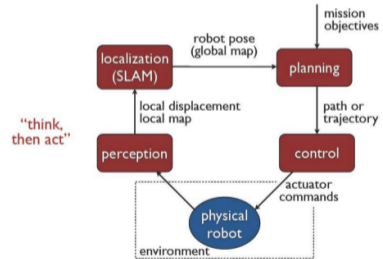
```
std_msgs/Header header
uint32 seq
time stamp
string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Simulation of a mobile robot

- Clone the package from the course organization `$ git clone https://github.com/RoboticsLab2023/r1_fra2mo_description.git`
- Two launch files:
 - One is to visualize the robot in RViz with RobotModel
`$ roslaunch r1_fra2mo_description display_fra2mo.launch`
 - The other is to spawn the robot in Gazebo
`$ roslaunch r1_fra2mo_description spawn_fra2mo_gazebo.launch`
- To move the mobile robot you can publish on the topic `/cmd_vel`
 - `$ rostopic pub --once /cmd_vel ...`
 - `$ rosrun rqt_robot_steering rqt_robot_steering`
 - Other packages to teleoperate a robot
`$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py`
- Hint: to have a better look of the TF tree you can use:
`$ rosrun rqt_tf_tree rqt_tf_tree`

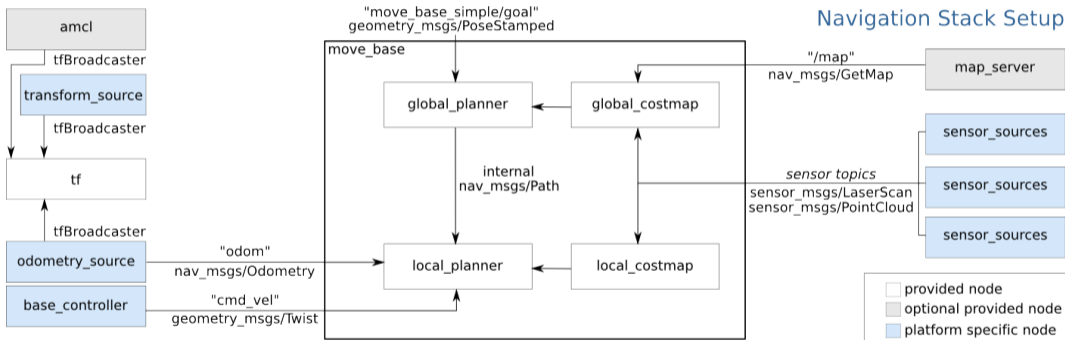
General architecture

- Any robotic task follows the paradigm "think then act"
- Also for autonomous navigation it works
- We have to define the blocks related to the perception, localization, and planning
- In ROS there is a package that already implements this logic, it's called `navigation_stack` \$ `sudo apt-get install ros-<DISTRO>-navigation`



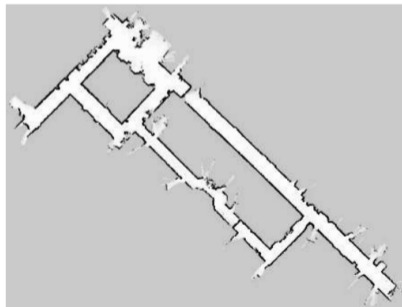
navigation_stack - Description

- It takes in information from odometry and sensor streams and outputs velocity commands to send to a mobile base
- A pre-requisite to use the navigation_stack is to have a tf transform tree
- The major component of the navigation_stack is move_base ROS Node
`$ sudo apt-get install ros-<DISTR0>-move-base`
- The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot



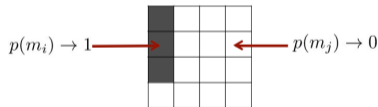
Grid map

- Discretize the world into cells
- Each cell is assumed to be occupied or freespace
- Large maps require substantial memory resources
- Do not rely on a feature detector
- Grid structure is rigid



Grid map

- Each cell is a binary random variable that models the occupancy
 - occupied cell: $p(m_i) = 1$
 - not-occupied cell: $p(m_i) = 0$
 - no knowledge: $p(m_i) = 0.5$
- Often the world is assumed **static**
- The cells are independent of each other
- The last assumption is fundamental, we can compute the probability distribution of a map as the product of the individual cells



$$p(m) = \prod_i p(m_i)$$

Grid map - example

m1	m2
m3	m4



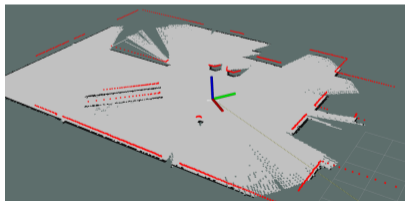
0.9	0.5
0.8	0.1

$$p(m) = \prod_{i=1}^4 p(m_i)$$

$$p(m) = 0.9 * (1 - 0.5) * 0.8 * (1 - 0.1) = 0.324$$

Gmapping

- OpenSlam's Gmapping is a laser-based SLAM
- It creates a 2-D occupancy map from the laser and poses data collected by a mobile robot
- The localization output is the transformation between the frame map and odom
- Based on a Rao-Blackwellized particle filter
- Developed almost 20 years ago!
- For more details see Gmapping



Gmapping - bringup

- The launch file should be composed by 4 sections of parameters:
 - general parameters
 - optimizers parameters
 - odometry parameters
 - particle filter parameters

```
<param name="base_frame" value="
    base_footprint"/>
<param name="odom_frame" value="odom"/>
<param name="map_frame" value="map"/>
<param name="map_update_interval" value="2.0
    "/>
<param name="maxUrange" value="6.0"/>
<param name="maxRange" value="8.0"/>
<param name="xmin" value="-40.0"/>
<param name="ymin" value="-40.0"/>
<param name="xmax" value="40.0"/>
<param name="ymax" value="40.0"/>
<param name="delta" value="0.05"/>
```

Gmapping - bringup

```
<!--Optimizer params-->
<param name="sigma" value="0.05"/>
<param name="kernelSize" value="1"/>
<param name="lstep" value="0.05"/>
<param name="astep" value="0.05"/>
<param name="iterations" value="5"/>
<param name="lsigma" value="0.075"/>
<param name="ogain" value="10.0"/>
<param name="lskip" value="0"/>
<param name="minimumScore" value="
  140"/>
```

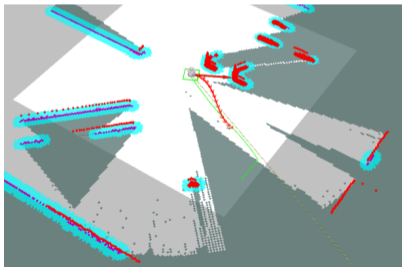
```
<!--Particle filter params -->
<param name="resampleThreshold"
  value="0.5"/>
<param name="particles" value="80"/>
<!--Likelihood params-->
<param name="llsamplerange" value="
  0.01"/>
<param name="llsamplestep" value="
  0.01"/>
<param name="lasamplerange" value="
  0.005"/>
<param name="lasamplestep" value="
  0.005"/>
```

Gmapping - bringup

```
<!--Odometry errors-->
<param name="srr" value="0.01"/>
<param name="srt" value="0.02"/>
<param name="str" value="0.01"/>
<param name="stt" value="0.02"/>
<param name="linearUpdate" value="0.5"/> <!--Process a scan each time the
  robot translates this far-->
<param name="angularUpdate" value="0.436"/> <!--Process a scan each time
  the robot rotates this far-->
<param name="temporalUpdate" value="-1.0"/> <!--Process a scan if the last
  scan processed is older than the update time in seconds. A value less
  than zero will turn time based updates off-->
```

Teb local planner

- This package implements an online optimal local trajectory planner for navigation and control of mobile robots as a plugin for the ROS navigation package
- The initial trajectory generated by a global planner is optimized during runtime
- The current implementation complies with the kinematics of differential-drive robots



Teb local planner - Description

- Teb needs as input:
 - An odometry message from the mobile robot
 - An ObstacleArrayMsg (optional if you will provide it a costmap), for further information about the message here
- And it provides as output:
 - `<name>/global_plan` (nav_msgs/Path)
 - `<name>/local_plan` (nav_msgs/Path)
 - `<name>/teb_poses` (geometry_msgs/PoseArray)

Teb local planner - Parameters

- Robot configuration parameters:
 - dynamic constraints of the planner
 - footprint model

```
# Robot
max_vel_x: 0.6 #0.1
max_vel_x_backwards: 0.3
max_vel_theta: 0.3 #15deg more or
less
acc_lim_x: 0.1 #was 1.0
acc_lim_theta: 0.3 #was 0.3
min_turning_radius: 0.0
footprint_model:
  type: "polygon"
  vertices: [[0.2, -0.2],
             [-0.2, -0.2],
             [-0.2, 0.2],
             [0.2, 0.2]]
```

Teb local planner - Parameters

- Frames parameters
- Trajectory parameters

```
odom_topic: odom
map_frame: map

# Trajectory
teb_autosize: True
dt_ref: 0.5
dt_hysteresis: 0.2
global_plan_overwrite_orientation:
  True
max_global_plan_lookahead_dist: 2.0
feasibility_check_no_poses: 5
publish_feedback: true
```


Teb local planner - Parameters

- Goal tolerance
- Obstacles

```
# GoalTolerance
xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.15
free_goal_vel: False

# Obstacles
min_obstacle_dist: 0.20
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.0
obstacle_poses_affected: 20
costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5
inflation_dist: 0.2
include_dynamic_obstacles: false
```

Teb local planner - Parameters

- Optimization
- For further information about parameters you can go here

```
# Optimization
no_inner_iterations: 3
no_outer_iterations: 2
optimization_activate: True
optimization_verbose: false
penalty_epsilon: 0.04
weight_max_vel_x: 2
weight_max_vel_theta: 1
weight_acc_lim_x: 1
weight_acc_lim_theta: 1
weight_kinematics_nh: 1000
weight_kinematics_forward_drive: 200.0
weight_kinematics_turning_radius: 1
weight_optimaltime: 1
weight_obstacle: 100
```

move_base

- This package will be the connector between the planner and the perception
- Its main task is to take a goal position as input and to publish the right velocities in `/cmd_vel`
- The goal is sent using an Action topic:
`move_base/goal` (`move_base_msgs/MoveBaseActionGoal`)
- In general the commands to `move_base` are sent using **actionlib**
- To set up `move_base` there are different configuration files to create, one for the generic `move_base`, two both for the local and global costmap and one for the common parameters.

move_base - Set up

[move_base_params.yaml](#)

```
max_planning_retries: 5
planner_patience: 10
number_of_recovery_behaviors: 3
planner_frequency: 5
controller_frequency: 10
allow_unknown: false
controller_patience: 3.0
oscillation_timeout: 10.0
oscillation_distance: 0.2
```

[costmap_common_params.yaml](#)

```
publish_voxel_map: false
transform_tolerance: 0.5
meter_scoring: true
obstacle_range: 7.0
raytrace_range: 8.0
footprint: [[0.2, -0.2],
            [-0.2, -0.2],
            [-0.2, 0.2],
            [0.2, 0.2]]
```

move_base - Set up

local_costmap_params.yaml

```
local_costmap:  
  global_frame: map  
  robot_base_frame: base_footprint  
  update_frequency: 5.0  
  publish_frequency: 10.0  
  static_map: false  
  rolling_window: true  
  width: 7.0  
  height: 7.0  
  resolution: 0.03
```

global_costmap_params.yaml

```
global_costmap:  
  global_frame: map  
  robot_base_frame: base_footprint  
  update_frequency: 5.0  
  publish_frequency: 2.0  
  rolling_window: false  
  resolution: 0.05  
  width: 30  
  height: 30  
  origin_x: -15  
  origin_y: -15
```

move_base - Set up

`local_costmap_plugin.yaml`

```
obstacles_layer:
  observation_sources: laser_scan_sensor
  laser_scan_sensor: {sensor_frame: laser_frame, data_type: LaserScan,
    topic: laser/scan, marking: true, clearing: true, min_obstacle_height:
    0.0, max_obstacle_height: 5.0, obstacle_range: 6.0, raytrace_range: 7.0,
    inf_is_valid: true}
inflater_layer:
  inflation_radius: 0.4
  cost_scaling_factor: 10.0
  lethal_cost_threshold: 200
plugins:
- {name: obstacles_layer, type: "costmap_2d::VoxelLayer"}
```

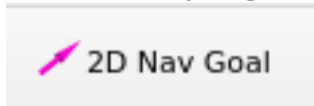
move_base - Set up

[global_costmap_plugin.yaml](#)

```
obstacles_layer_g:
  observation_sources: laser_scan_sensor
  laser_scan_sensor: {sensor_frame: laser_frame, data_type: LaserScan,
    topic: laser/scan, marking: true, clearing: true}
inflater_layer_g:
  inflation_radius: 0.2
  cost_scaling_factor: 10.0
  lethal_cost_threshold: 200
plugins:
- {name: static_map, type: "costmap_2d::StaticLayer"}
- {name: inflater_layer_g, type: "costmap_2d::InflationLayer"}
recovery_behaviors:
- name: conservative_reset
  type: clear_costmap_recovery/ClearCostmapRecovery
```

move_base - actionlib

- Once move_base is correctly initialized we have to send to it the goal position
- The easiest way to give an input to move_base is to send the goal through RViz

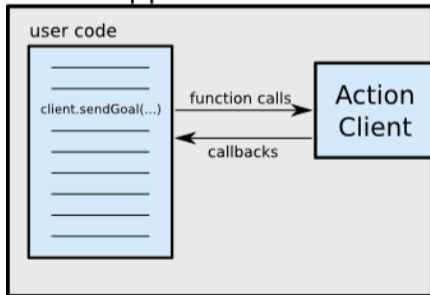


- With this button you can send a 2D pose to move_base through the topic move_base_simple/goal
- The message type is a geometry_msgs/PoseStamped
- If you want to avoid this use of this button, the recommended way to send goal to move_base is by using the SimpleActionClient
- In this way we can be able to track the status of the goal

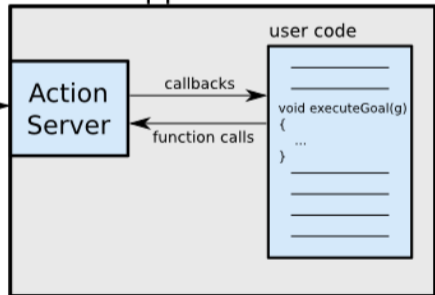
Actionlib - Client and Server interaction

- The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks

Client Application



Server Application



ROS

Actionlib - Client and Server interaction

In order for the client and server to communicate, we need to define a few messages on which they communicate:

- **Goal:** to accomplish tasks using actions, a goal can be sent to an ActionServer by an ActionClient. In the case of moving the base, the goal would be a PoseStamped message that contains information about where the robot should move to in the world.
- **Feedback:** feedback provides server implementers a way to tell an ActionClient about the incremental progress of a goal. For moving the base, this might be the robot's current pose along the path.
- **Result:** a result is sent from the ActionServer to the ActionClient upon completion of the goal. This is different than feedback since it is sent exactly once. This is extremely useful when the purpose of the action is to provide some sort of information. For move_base, the result isn't very important, but it might contain the final pose of the robot.

move_base - SimpleActionClient

- Let's write a C++ program that sends one goal using the actionlib, these are the important steps
- First we need to include the right libraries:

```
include <move_base_msgs/MoveBaseAction.h>  
include <actionlib/client/simple_action_client.h>
```

- We define a convenience typedef for a SimpleActionClient that will allow us to communicate with actions that adhere to the MoveBaseAction action interface:

```
typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
```

move_base - SimpleActionClient

- Construct an action client that we'll use to communicate with the action named "move_base" that adheres to the MoveBaseAction interface:

```
MoveBaseClient ac("move_base", true);
```

- Wait for the action server to report that it has come up and is ready to begin processing goals.

```
while(!ac.waitForServer(ros::Duration(5.0))){  
  ROS_INFO("Waiting for the move_base action server to come up");  
}
```

move_base - SimpleActionClient

- Then create a goal to send to move_base using the `move_base_msgs::MoveBaseGoal` message type which is included automatically with the `MoveBaseAction.h` header:

```
move_base_msgs::MoveBaseGoal goal;
//we'll send a goal to the robot to move 1 meter forward
goal.target_pose.header.frame_id = "base_link";
goal.target_pose.header.stamp = ros::Time::now();
goal.target_pose.pose.position.x = 1.0;
goal.target_pose.pose.orientation.w = 1.0;
ROS_INFO("Sending goal");
ac.sendGoal(goal);
```

move_base - SimpleActionClient

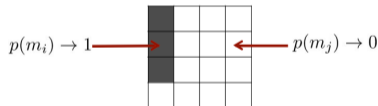
- The only thing left to do now is to wait for the goal to finish using the `ac.waitForGoalToFinish` call which will block until the `move_base` action is done processing the goal we sent it

```
ac.waitForResult();  
  
if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)  
    ROS_INFO("Hooray, the base moved 1 meter forward");  
else  
    ROS_INFO("The base failed to move forward 1 meter for some reason");
```

Autonomous Navigation

Grid map

- Each cell is a binary random variable that models the occupancy
 - occupied cell: $p(m_i) = 1$
 - not-occupied cell: $p(m_i) = 0$
 - no knowledge: $p(m_i) = 0.5$
- Often the world is assumed **static**
- The cells are independent of each other
- The last assumption is fundamental, we can compute the probability distribution of a map as the product of the individual cells



$$p(m) = \prod_i p(m_i)$$

Grid map - Inverse range sensor model

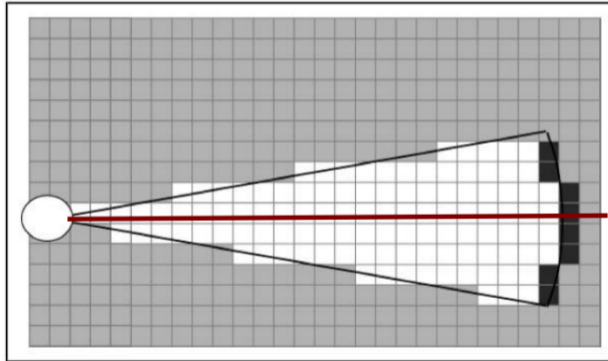


Figure: from book *Probabilistic Robotics*, Thrun S., Burgard W., Fox D.

Grid map - Inverse range sensor model

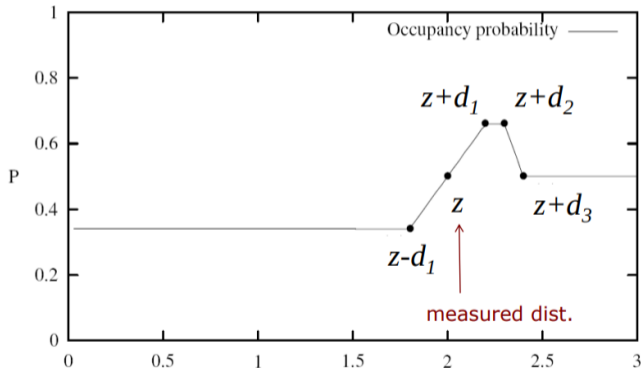


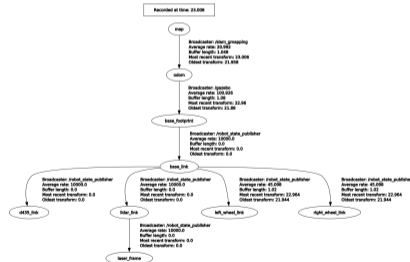
Figure: from book *Probabilistic Robotics*, Thrun S., Burgard W., Fox D.

TF listener - Computing robot pose

- In a project of autonomous exploration there is the need to know the pose of the robot.
- So far we manage the odometry message as the current pose of the robot.
- In real-world applications this method could be very inaccurate due to unmodeled effects that affect the computation of the odometry such as wheel slip, imperfections in the ground, inaccuracy of encoders, and limitation of the odometry model.
- To overcome this problem, we integrated in our navigation stack a SLAM algorithm (Gmapping) which, building a map of the environment, can simultaneously refine the robot pose.

TF listener - Computing robot pose

- The fundamental requirements for move_base and Gmapping is that the robot has the tf transform tree
- Gmapping publishes a periodic update of the transformation between the frame map and odom, while the transformation between the base_footprint and odom is published by the differential_drive_controller_plugin in simulation
- To retrieve the robot pose we have to compute the transformation between the frame map (fixed) and the base_footprint



Tf listener - Code

- It is not possible to directly subscribe to the /tf topic
- We have to define a TF listener:
- The tf package provides an implementation of a TransformListener to help make the task of receiving transforms easier

```
#include <tf/transform_listener.h>
```

- We create a TransformListener object. Once the listener is created, it starts receiving tf transformations over the wire, and buffers them for up to 10 seconds

```
tf::TransformListener listener;
```

TF listener - Code

- We query the listener for a specific transformation:

```
try {  
    listener.waitForTransform(destination_frame, original_frame, ros::Time(0), ros::Duration  
        (10.0) );  
    listener.lookupTransform(destination_frame, original_frame, ros::Time(0), transform);  
} catch (tf::TransformException ex) {  
    ROS_ERROR("%s", ex.what());  
}
```

- Let's take a look at the four arguments:
 - We want the transform from frame /original_frame to /destination_frame.
 - The time at which we want to transform. Providing ros::Time(0) will just get us the latest available transform.
 - The object in which we store the resulting transform.



TF listener - Code

- The `waitForTransform()` takes four arguments:
 - Wait for the transform from this frame...
 - ... to this frame,
 - at this time, and
 - timeout: don't wait for longer than this maximum duration.
- This function will actually **block** until the transform between the two frames becomes available OR until the timeout has been reached.

SSH protocol

- Secure SHell (SSH) is a protocol that allows you to securely connect to a remote computer.

```
$ sudo apt install openssh-client
```

- The client and server must be connected to the same net
- To connect to the remote computer use this command:

```
$ sudo ssh remote_name@192.168.X.X
```

- To connect we should know the name of the remote computer, the password, and the IP address
- Once the connection is done, a terminal on the remote computer will appear

ROS across multiple machines

- To pass files to the remote computer we can use `git clone` or softwares which exploit the SSH protocol to exchange files.
- With mobile robots is important to have feedback of what is happening on the robot. ROS is a distributed computing environment, any node may need to communicate with any other node, at any time.
- We want to configure ROS on both the robot and the ground station:
 - We only need one master. Select one machine to run it on.
 - All nodes must be configured to use the same master, via `ROS_MASTER_URI`
 - There must be complete, bi-directional connectivity between all pairs of machines, on all ports

ROS across multiple machines

- We have to run these commands on both machines:

```
export ROS_IP=ip_address  
export ROS_MASTER_URI=http://ip_address:11311
```

- These commands can be included in the `bashrc` file in order to be executed at every opening of a terminal.
- We have to define where the master goes defining the `ip_address` of the `ROS_MASTER_URI` inserting one of the two machines IP.
- As `ROS_IP` we have to put the one of each machine