Contents

CHAPTER 1

Introduction to ROS and its Package Management

ROS

roscd ⇒ changes the directory to ↑ catkin_ws Workspace.

roscore ⇒ Runs the ↑ ROS Master node

rosnode list ⇒ list all the ROS Node

rostopic list ⇒ " " " topics

rosrun Packagename {eg-turtlesim} {To run a ros node} tab 2x

example rosrun turtlesim turtlesim_node

rostopic echo topicname {echo's what is being published in the topic}

Introduction to ROS and its Package Management {Mastering ROS for Robotics Programming}

⇒ The basic building blocks of the ROS software framework are ROS packages

↳ Create a Wiki page for our package on the ROS website to contribute to the ROS community.

★ Why should we learn ROS?

Robot application development platform that provides various features such as:-

- Message passing
- Distributed Computing
- Code reusing etc...

⇒ ROS comes with ready to use capabilities, for example, SLAM & AMCL.

{Simultaneous Localization and Mapping} {Adaptive Monte Carlo Localization}

MoveIt → {Package for motion planning of robot manipulator}

⇒ ROS is packed with tons of tools for debugging (rqt_gui), Visualizing (RViz) and Performing Simulation (Gazebo).

⇒ ROS is packed with device drivers and interface Packages for various sensors & actuators in Robotics.

⇒ The ROS message-passing middleware allows Communicating between different nodes. These nodes can be programmed in any language that has Ros client libraries. (C, C++, Python, Java)

⇒ ROS can easily handle Concurrent resource.

⇒ ROS has Active Community.

★ Reasons for not Preferring ROS for Robots

⇒ It is difficult to learn ROS.

⇒ To get started with Gazebo is not an easy task.

⇒ Difficulties in robot modeling.

↳ Robot modeling in ROS is performed using URDF, which is an XML based robot description.

↳ There is a SolidWorks plugin to convert a 3D model from SolidWorks to URDF.

⇒ We always need a computer to run ROS.

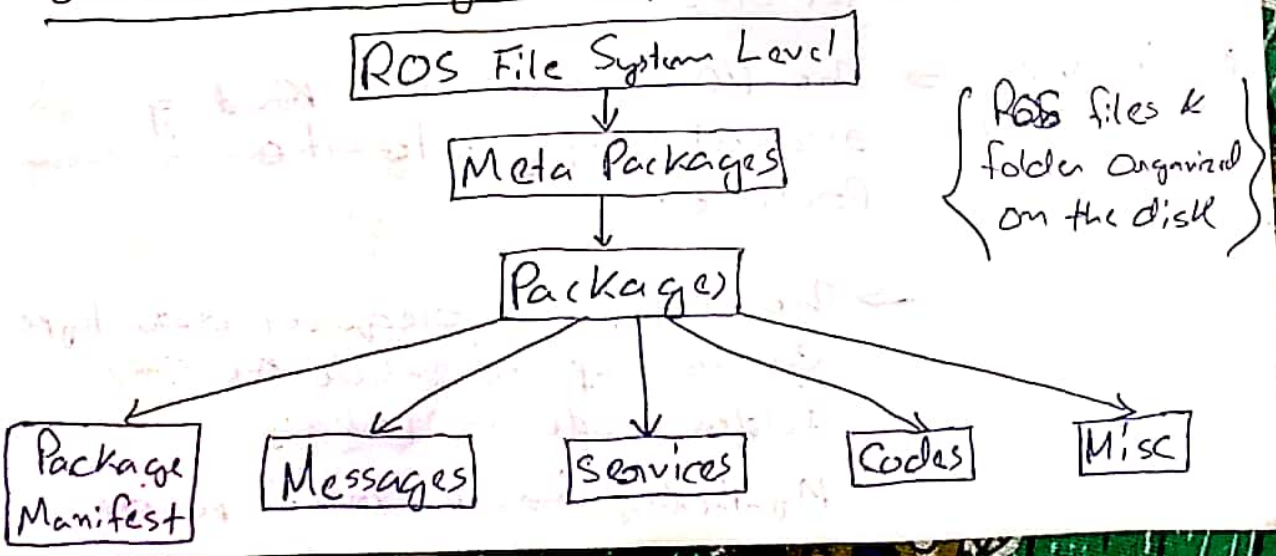
↳ Small robots that work completely on microcontroller don't require a ROS system.

↳ ROS is only required when we want to perform high-level functionalities such as autonomous navigation and motion planning.

⇒ When we deploy ROS on a commercial product, a lot of things need to be taken care of: One thing is <u>Code quality</u>.

↳ Code quality is a loose approximation of how long-term useful and long-term maintainable the code is.

★ Understanding the ROS file System level



Packages ⇒ It contains ROS runtime process (nodes), libraries, configuration files & so on! which are organized together as a single unit.

Package manifest ⇒ It is inside a package that contains information about the Package, author, licence, dependencies, compilation flags and so on.

(Package.xml file)

Meta package ⇒ Term meta package is used for a group of package for a special purpose.

Meta Package manifest ⇒ Similar to package manifest, difference are that it might include Packages inside it as runtime dependencies and declare an export tag.

Messages (.msg) ⇒ The ROS messages are a type of information that is sent form one ROS process to the other.

Services (.srv) ⇒ The ROS service is a kind of request/reply interaction between Processes.

→ The .reply and request data type can be defined inside the srv folder inside the package.

My_Package/srv/MyServiceType.srv

Repositories ⇒ Most of the ROS packages are maintained using a Version Control System (VCS) such as Git.

⇒ The package in the repositories can be released using a Catkin release automation tool called bloom.

★ ROS packages

Commands to Create, modify and work with the ROS packages.

Catkin_Create_pkg ⇒ This command will Create new package.

rospack ⇒ This Command is used to get information about the package in the file System.

Catkin_make ⇒ This command is used to build the Package in the workspace.

rosdep ⇒ This Command will install the System dependencies required for this package.

To work with packages, ROS provides a bash-like command called **rosbash**, which is used to navigate and manipulate the ROS packages. Some of the rosbash Commands:

roscd ⇒ This Command is used to change the Package folder.

roscp ⇒ This command is used to Copy a file from a package.

rosed ⇒ This command is used to edita file.

rosrun ⇒ This Command is used to run an executable inside a package.

* ROS Meta packages

⇒ Specialized package in ROS that only Contain one file, that is a package.xml file.

⇒ Meta package simply group a set of multiple Packages as a single logical package.

* Understanding the ROS Computation graph level

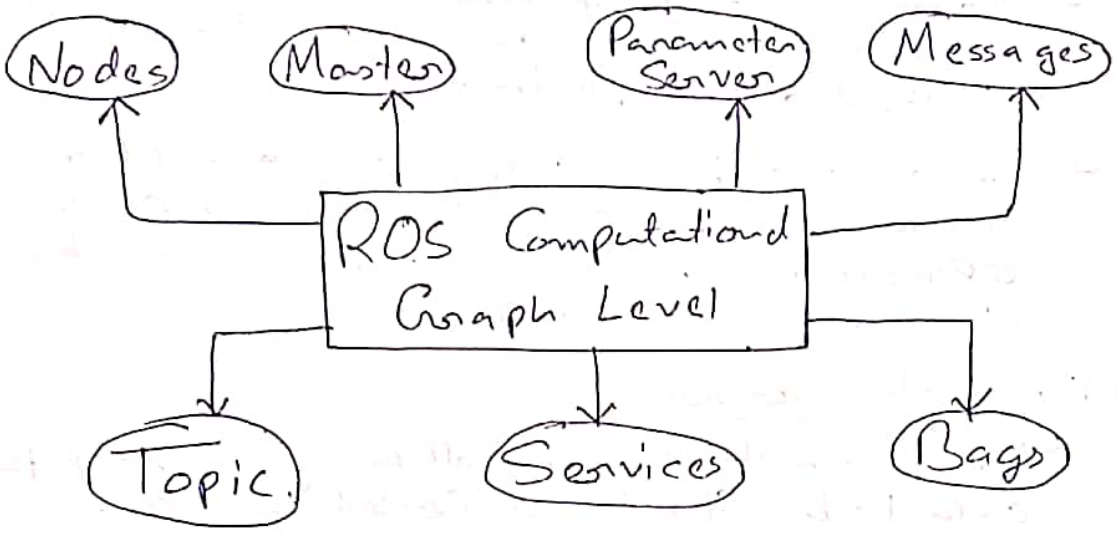
⇒ The Computation in ROS is done using a network of process called ROS nodes.

⇒ The main Concepts in the Computation graph are ROS Nodes, Master, Parameter Server, Messages, Topic, Services and Bags.

⇒ The ROS Communication related packages including Core client libraries, such as rospy and roscpp and the implementation of concepts such as topics, nodes, Parameters and Services are included in a stack called ros_comm.

↳ This stack also consists of tools such as rostopics, rosparam, rosservice and rosnode.

⇒ Ros_Comm stack contains the ROS communication middleware packages and these packages are collectively called ROS Graph layer.



Nodes

→ Nodes are the process that perform Computation.

→ Each ROS node is written using ROS client libraries such as roscpp and rospy.

→ In a robot, there will be many nodes to Perform different kinds of tasks.

→ One of the aim of ROS nodes is to build simple Processes rather than a large process with all functionality.

Master

→ The ROS Master provides name registration and lookup to the rest of the nodes.

→ Nodes will not be able to find each other, exchange messages or invoke services without a ROS Master.

→ In a distributed System, we should run the master on one Computer and other remote nodes can find each other by Communicating with the master

Parameter Server

→ The parameter Server allows you to keep the data to be Stored in a Central location.

→ All nodes can asses and modify these values

→ Parameter Server is a part of ROS Master.

Messages

→ Nodes Communicate with each other using messages.

Topics

→ Each message in ROS is transported using named buses called topics.

→ When a node Sends a message through a topic, then we can say that node is publishing a topic.

→ When a node receives a message through a topic, then we can say that the node is Subscribing to a topic.

→ The publishing node and Subscribing node are not aware of each other's existence.

→ Each topic has a unique name, and any node can access this topic and send data through it as long as they have the right message type.

Services

→ In some robot applications, Publish/Subscribe model will not be enough if it needs to request/response interaction.

→ ROS services are used in these case.
  ↳ We can define a service definition that contains two part:
    → Requests
    → Responses.

→ Using ROS services, we can write a server node and client node.

Bags

→ Bags are a format for saving and playing back ROS message data.

→ Bags are very useful features when we work with complex robot mechanisms.

★ Understanding ROS Nodes

⇒ ROS nodes are a process that perform computation using ROS client libraries such as roscpp and rospy.

⇒ One node communicate with other nodes using ROS Topics, Services & Parameters.

⇒ All running nodes should have a name assigned to identify them from the rest of the system.

⇒ There is a rosbash tool to introspect ROS nodes :-

rosnode info [node_name]
{This will print info about the node}

rosnode kill [node_name]
{This will kill a running node}

rosnode list
{This will list all the running nodes}

rosnode machine [machine_name]
{This will list the nodes running on a Particular machine or a list of machine}

rosnode ping
{This will check the connectivity of a node}

rosnode cleanup
{This will purge the registration of unreachable nodes}

* ROS messages

⇒ ROS nodes communicate with each other by Publishing message to a topic.

⇒ Message are a simple data structure containing field types.

⇒ ROS has inbuilt tools called rosmsg to get information about ROS messages :-

rosmsg show [message]
{This shows the message description}

rosmsg list
{This lists all message}

rosmsg md5 [message]
{This displays md5 sum of a message}

{MD5 checksum comparision is used to confirm whether the publisher and subscriber exchange the same message data types.}

rosmsg Package [Package_name]
{This lists message in a Package}

rosmsg Packages [Package_1] [Package_2]
{This lists packages that Contain messages}

* ROS topics

⇒ ROS topics are named buses in which ROS nodes exchange messages.

⇒ ROS nodes are not interested to know which node is publishing the topic or subscribing topics, it only looks for the topic name and whether the message types of publisher and subscriber are matching.

⇒ ROS node communicate with topics using TCP/IP based transport known as TCPROS. (Default)

↳ Another type is UDPROS.

⇒ The rostopic command can be used to get information about ROS topics:-

rostopic bw /topic
{Displays the bandwidth used by a given topic}

rostopic echo /topic
{Prints the content of a given topic}

rostopic find /message-type
{This command will find topics using the given message type}

rostopic info /topic
{Prints information about the topic}

rostopic list
{lists all the active topics in ROS systems}

rostopic pub /topic message_type args

{This Command is used to publish a value to a topic with a message type}

rostopic type /topic

{This will display type of message for the given topic}

★ ROS Services

⇒ When we need a request/response kind of communication in ROS, we have to use the ROS services:

↳ Mainly used in distributed systems.

⇒ We have to define a request datatype and a response datatype in a srv file.

rosservice call /service args

{This tool will Call the service using the given arguments}

rosservice find service_type

{find services in the given service type}

rosservice info /services

{This will print information about the given service}

ros service list

{list active service running on the system}

rosservice type /service

{This Command will print the service type of the given service}

rosservice uri /service

{this tool will print the service ROSRPC URI}

* ROS bags (.bag)

⇒ A bag file in ROS is for storing ROS message data from topics and services.

⇒ Bag files are created using the rosbag command, which will subscribe one or more topics & store the message's data in a file as it's received.

⇒ The main application of rosbag is data logging. The robot data can be logged and can visualize and process offline.

rosbag record [topic_1] [topic_2] -O [bag_name]

{This command will record the given topic into a bag file}

rosbag play [bag_name]

{This will playback the existing bag file}

★ Understanding ROS Master

⇒ ROS Master is much like DNS server

↓

(Domain name System)

⇒ When any node starts in the ROS System, it will start looking for ROS Master and register the name of the node in it.

↳ Master has the details of all nodes currently running on the ROS System.

★ Using the ROS Parameter

⇒ While programming a robot, we might have to define robot parameters such as robot Controller gain such as P, I and D.

⇒ ROS provides a parameter Server, which is a shared server in which all ROS nodes can access parameters from this Server.

↳ A node can read, write, modify and delete Parameter values from the parameter Server.

⇒ The rosparam tool used to get and set the ROS parameter from the Command line :-

rosparam set [Parameter_name] [value]

{To set a value in the given parameter}

rosparam get [Parameter_name]

{To retrieve a value from the given parameter}

rosparam load [YAML file]

{ROS parameter can be saved into a YAML file and it can load to the parameter server using this command}

rosparam dump [YAML file]

{Dump existing ROS parameters to a YAML file}

rosparam delete [Parameter_Name]

{To delete a given Parameter}

rosparam list

{List the existing parameter name}

★ Understanding ROS Community level

- **Distributions**
  - ↳ ROS distributions are a collection of versioned meta package that we can install.
- **Repositories**
  - ↳ Where different institutions can develop and release their own robot software components.
- **ROS Wiki**
  - ↳ Main forum for documenting information about ROS
- **ROS Answer**
  - ↳ To ask questions related to ROS
- **Blog**
  - ↳ The ROS blog updates with news, Photo, and Videos related to the ROS Community.

★ Creating a new package in ROS

⇒ Packages are created is Src folder of Catkin workspace (Catkin_ws)

① Catkin_Create_Pkg Packagename dependencies

{eg ROS_Tutorial} { std_msgs, rospy, roscpp }

② go to Catkinwork directory

③ Catkin_make. {To Compile the workspace}

★ To run a node

rosrun Packagename nodename.

CHAPTER 2

ROS Official Tutorial

ROS Tutorials {Beginner Level}

1. Core ROS Tutorials

1.1) Beginner Level

① Installing and Configuring your ROS Environment

source /opt/ros/melodic/setup.bash

⇒ You will need to run this command on every new shell you open to have access to the ROS commands, Unless you add this line to your .bashrc.

* Create a ROS Workspace

① mkdir -p ~/catkin_ws/src

② cd ~/catkin_ws/

③ catkin_make {for creating the workspace}

⇒ Inside catkin_ws there will be a 'devel' folder which contain setup.bash.

↓

{Sourcing this file will overlay this workspace on top of your environment}

② Navigating the ROS Filesystem

`rospack find [Package_Name]`

{This will return location of that package}

`roscd [Package_Name]`

{Changes the directory to that package}

`roscd`

{change the directory to the current workspace}

`rosls [Package_Name]`

{this lets you ls directly in a package by name rather than absolute path}

⇒ Pressing `tab` gives the valid option list.

③ Creating a Package (#1) {31}

⇒ For a package to be considered a catkin Package it must meet few requirements:

1) The package must contain a catkin compliant `package.xml` file.
   → {Provides meta info about Package}
2) The package must contain a `CMakeLists.txt` which uses catkin.
3) Each package must have its own folder.

★ Customizing Your Package

ⓐ description tag

<description> Description of the Package </description>

ⓑ Maintainer tag

<maintainer email="Your@yourdomain.tld">
Yourname </maintainer>

ⓒ License tags

<license> BSD </license>

ⓓ dependencies tags

⇒ The dependencies are split into: ~~Build~~

- buildtool_depend { catkin }
- build_depend
- build_export_depend
- exec_depend

④ Building a ROS Package

Catkin_make

↑ this Command is used to build the package

⑤ Understanding ROS Nodes

⇒ When you open a new terminal your environment is reset and your ~/.bashrc file is sourced.

`rosrun [Packagename] [node_name]`

{To run a node within a package}

⇒ To change the name of node

`rosrun [Package_name] [Old:Node_name] _name:=[New_node_Name]`

⑥ Understanding ROS Topic

* Using rqt_graph

⇒ rqt_graph creates a dynamic graph of what's going on in the system.

→ rqt_graph is part of the rqt package.

`rosrun rqt_graph rqt_graph`

{To run rqt_graph}

`[ros command] -h`

{To get details about the Ros command}

`rostopic echo [topic_name]`

{To display message being passed through the topic}

ros topic type [topic_name]

{Returns the message type of any topic being published}

* Using ros topic pub

↳ ros topic pub publishes data on to a topic currently advertised.

ros topic pub [topic] [msg_type] [args]

-r ⇒ To publish steady stream of command

-1 ⇒ to only publish one message then exit

ros topic hz [topic_name]

{Reports the rate at which data is being published}

⇒ rqt_Plot displays a scrolling time plot of the data published on topic.

ros run rqt_plot rqt_plot

{TO run rqt_plot}

{In this GUI you can enter the topic variable to plot}

⑦ Understanding ROS Services & Parameters

rosservice list

{lists all the Services}

rosservice type [service name]

{return type of service}

rosservice call [Service] [args]

{To call a Service}

eg ⇒ rosservice call /clean

{Clears the background of turtle sim_node}

Example

{rosservice type /spawn | rossrv show}

* **rosparam**

⇒ Allows you to store and manipulate data on ROS parameter server.

⇒ rosparam uses YAML markup language for Syntax.

rosparam get /

{To show us the contents of the entire parameter Server}

⑧ Using rqt_Console and roslaunch

rqt_Console

↳ It attaches to ROS's logging framework to display output from nodes.

rqt_logger_level

↳ Allows us to change the verbosity level (DEBUG, WARN, INFO and ERROR) of nodes as they run

```
rosrun rqt_Console rqt_Console
```
{To run rqt_Console}

```
rosrun rqt_logger_level rqt_logger_level
```
{To run rqt_logger_level}

⇒ Logging level are prioritized in the following order:-

Fatal → Error → Warn → Info → Debug

⇒ By setting the logger level, you will get all messages of that priority level or higher.

★ Using roslaunch

```
roslaunch [Package_name] [filename.launch]
```
{To start a launch file}

⑨ Using rosed to edit files in ROS

rosed Package_name filename

{To edit the file in the package}

⇒ Default editor of rosed is Vim.

If it is not then then,

export EDITOR = 'nano -w'

⑩ Creating a ROS message (#2) (35) and SRV

* Msg ⇒ .msg files are simple text files that describe the fields of a ROS message.
  → They are used to generate source code for messages in different languages.

* Srv ⇒ An .srv file describes a service. It is composed of two parts: a request & a response.

msg
↳ message_name.msg

⇒ edit Package.xml & CMakeLists.txt accordingly.

Srv
↳ Service_name.srv

⇒ edit Package.xml & CMakeLists.txt accordingly.

(11) Writing a Simple Publisher and Subscriber (C++)

★ Writing the Publisher Node

```
#include "ros/ros.h"          → ros.h is in ros package
#include "std_msgs/String.h"  → String.h in Std_msgs Package
#include <sstream>
```

StringStream

⇒ used to convert int into String and String into int

```
StringStream sso;   → sso>>
                    → sso>>
int a = 55;
sso << a; // Output stored in StringStream sso from a
String b;
sso >> b; // String stored in b from sso.
```

⇒ A StringStream associates a String object with a stream allowing you to read from the string as if it were a stream (like Cin)

Basic methods are:-

① Clear() ⇒ to clear the stream

② str() ⇒ to get and set String object whose content is present in stream.

③ << ⇒ add a string to the StringStream object.

④ >> ⇒ read something from the StringStream object

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

argc → Number of things that we enter in Terminal while running our executable.

argv → It stores what we enter in the command line while running our executable.

argv[0] → first element

argv[1] → Second element (command)

and so on

```
ros::init(argc, argv, "talker");
```

ros:: → function init is in ros namespace

"talker" → Name of the node

The name used here must be a base name, i.e. it cannot have a / in it

```
ros::NodeHandle n;
```

NodeHandle is the main access point to communications with the ROS system

→ The first NodeHandle created will actually do the initialization of the node, and the last one destructed will cleanup any resources the node was using

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>
                              ("chatter", 1000);
```

⇒ The advertise( ) function is how you tell ROS that you want to publish on a given topic name.

⇒ This invokes a call to the ROS master node.

which keeps a registry of who is publishing & who is subscribing

⇒ After this master node will notify anyone who is trying to subscribe to this topic name, and they will in turn negotiate a peer-to-peer connection with this node.

⇒ advertise() returns a Publisher object which allows you to publish message on that topic through a call to publish().

⇒ Once all copies of the returned Publisher object are destroyed, the topic will be automatically unadvertised.

⇒ Std_msgs:: String is msg type
⇒ Chatter is name of topic
⇒ 1000 ⇒ Size of buffer

ros::Rate loop_rate(10);

⇒ Rate object allows you to specify a frequency that you would like to loop at.
⇒ It will keep track of how long it has been since the last call to Rate::sleep(), and sleep for the correct amount of time.

```
int count=0;
while (ros::ok())
{
```

⇒ ros::ok() always returns true except if :-
(1) Ctrl+C is pressed
(2) We have been kicked off the network by another node with the same name.
(3) ros::shutdown() has been called by another part of the application.
(4) all ros::NodeHandles has been destroyed.

std_msgs::String msg

{message of name msg and type String in std_msgs name space}

```
std::stringstream ss;
ss << "hello world" << count;
msg.data = ss.str();
ROS_INFO("%s", msg.data.c_str());
```

⇒ ROS_INFO is replacement for Printf.

{ c_str returns a const char * that points to a null terminated string (i.e. C style string) }

```
chatter_pub.publish(msg);
ros::spinOnce();
```

{
⇒ Calling ros::spinOnce() here is not necessary.
⇒ If we were to add a subscription into this application, and did not have ros::spinOnce(), here your callbacks would never get called.
}

```
loop_rate.sleep();
++count;
}
return 0;
}
```

★ Writing the Subscriber Node

const char * ptr

char * const ptr

↓

Value it is pointing to can change but pointer cannot change

→ Value it is pointing to is Constant but Pointer can change address.

eg:
a = 5
const char * ptr = &a;
*ptr = 'b'; Not Valid
b = 6;
ptr = &b; Valid

#1 {ros} Creating a catkin Package

1) Go to src folder of your work space.
   {or change current working directory to src folder of your workspace}

2) `Catkin_Create_Pkg` <Package_Name> [Dependency 1] [Dependency 2] ....

Example
{std_msgs, rospy, roscpp}

building the package in Catkin workspace

1) change working directory to catkin_ws
2) `Catkin_make`

* Writing the Subscriber Node

```
#include "ros/ros.h"
#include "std_msgs/String.h"
void chattercallback (const std_msgs::String::ConstPtr& msg)
{
  ROSINFO ("I heard [%s]", msg->data.c_str());
}

int main (int argc, char **argv)
{
ros::init (argc, argv, "listener");
ros::NodeHandle handle;
ros::Subscriber sub;
sub = handle.subscribe ("chatter", 1000, chatterCallback);
ros::spin()
return 0
}
```

ros::spin() enters a loop, calling message callbacks as fast as possible.

add_exe
target_li
add_depe

⇒ Aft
and

* Building your Nodes

Add the Folling in the CMakeLists.txt

```
add_executable (talker src/talker.cpp)
target_link_libraries (talker ${catkin_LIBRARIES})
add_dependencies (talker test_Package1_generat
                  _message_cpp)
```

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

⇒ After that change pwd to your catkin workspace and execute catkin_make command.

12 Examining the simple publisher and Subscriber

rosrun PackageName nodeName

{To run the node}

⇒ Before running any node you need to run ros core.

13 Writing a Simple Service and Client

To create Service node which will receive two ints and return the sum.

Service Node

```
# include "ros/ros.h"
# include "test_Package1/AddTwoInts.h"
```

{Header file generated from the srv file.}

```
bool add (test_Package1::AddTwoInts::Request &req,
          test_Package1::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("Sending back response: [%ld]", (long int)res.sum);
```

(#1) 12.5)

Creating a ROS msg

* Creating a srv

1) Make a directory srv in

2) Add AddTwoInts.srv t

int64
int64
---
int64

3) In Package.xml

<build_depend> message_
<exec_depend> message

4) In CMakeLists.txt

→ Add message_gene
→ Add AddTwoInts.s
→ Add Std_msgs in

5) Catkin_make install

return true;
}

#2 (25)

Creating a ROS msg and Srv

* Creating a Srv

1) Make a directory srv in your package.

2) Add AddTwoInts.srv to it.

```
int64 a      } request
int64 b
---
int64 sum    } response
```

3) In Package.xml

<build_depend> message_generation </build_depend>

<exec_depend> message_runtime </exec_depend>

4) In CMakeLists.txt

- Add message_generation in find package( )
- Add AddTwoInts.srv in add_service_files()
- Add Std_msgs in 'generate_messages()

5) Catkin_make install

---

return true;

}

```
int) req.a, (long int) req.b );
(long int) res.sum);
```

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service;
  service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints");
  ros::spin();

  return 0;
}
```

{Service name}

Writing the Client Node

```
#include "ros/ros.h"
#include "test_package1/AddTwoInts.h"
#include <cstdlib> ——→ {For Converting string to integer}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");

  if (argc != 3)
  {
    ROS_INFO("Usage: add_two_ints_client X Y");
    return 1;
  }
```

```
ros::NodeHandle n;
ros::ServiceClient client
client = n.serviceClient<test_package1::AddTwoInts>
        ("add_two_ints");

test_package1::AddTwoInts srv;
srv.request.a = atoll (argv[1]);
srv.request.b = atoll (argv[2]);
if (client.call(srv))
{
ROS_INFO ("Sum: %ld", (long int) srv.response.sum);
}
else
{
ROS_ERROR("Failed to call service add_two_ints");
return 1;
}
return 0;
}
```

To Convert string to long long int

This actually calls the service with argument srv

Building the node

1) Declare the node in CMakeList.txt.
2) Catkin_make

Example

```
add_executable (add_two_ints_Server src/add_two_ints
                _Server.cpp)
target_link_libraries (add_two_ints_Server
                $${catkin_LIBRARIES})
add_dependencies (add_two_ints_Server
                ${test_package1_gencpp)
```

(17) Recording and playing data back

1. Make a directory named bagfiles.
2. rosbag record -a

   → This will record all the Published data in a bag file.

   { Bag file name will begin with year, date and time and the suffix .bag }

③ ros2bag info bagfilename

→ This will display all the information about the bag file.

④ ros2bag play bagfile_name.

→ This will play the bag file Publishing all the data in sequence

⑤ ros2bag record -O subset topic1 topic2

→ This will log to a file named subset.bag the data published by topic 1 and topic2

Note

⇒ ros2bag is limited in its ability to exactly duplicate the behaviour of a running system in terms of when messages are recorded and processed.

↳ Very sensitive to small changes in timing in the system.

(18) Getting Started wit roswtf tool

⇒ roswtf examines your system to try and find problems.

(19) Navigating the ROS wiki:

- → ROS Wiki: Landing Page
- → ROS Package pages
- → ROS Stack Pages

(20) Where Next?

ROS → (1) Robot middle-ware

ROS → (2) Standard mechanism for developers around the world to share their code.

{ To main reason to use ROS

1) Launching a Simulator
2) Exploring RViz
3) Understanding TF

⇒ The TF package transforms between different coordinate frames used by your robot and keeps track of these transforms over time.

⇒ Constructing a URDF model of your robot.

4) Going Deeper

→ actionlib

{This package provides a standardized interface for interacting with Preemptible task.}

→ Navigation

{2D navigation: map-building and path planning}

→ MoveIt

{To Control the arms of your robot}

Parameter Server

① getParam()

bool getParam (const std::string& key
, parameter_type& output_value) const

{Parameter name} — {Place to put retrieved data}

⇒ It returns bool, which provides the ability to check if retrieving the parameter succeeded or not.

② param()

NodeHandle.param<std::string> ("my_param", s, "default")

③ Setting Parameters

n.setParam ("my_param", "hello there");

④ Deleting Parameters

n.deleteParam ("my_param");

⑤ Searching for Parameters

N.searchParam ("b", param_name);

small

Ros Message

① Creating a msg

① Package Name
→ msg
→ messageName.msg

String first_name
String last_name
Uint8 age
Uint32 score

{ Example }

② In package.xml file

<build_depend> message_generation </build_depend>
<exec_depend> message_runtime </exec_depend>

③ In CMakeLists.txt file

⇒ Add message_generation in find package.

⇒ Add CATKIN_DEPENDS message_runtime in catkin_package.

⇒ Add messageName.msg in add_message_files

⇒ Add std_msgs in generate_messages.

Launch file

① Using roslaunch

↓

{It starts nodes as defined in a launch file}

roslaunch [Package] [filename.launch]

⇒ launch files are kept in launch directory inside the package.

② Launch file

→ It is a xml file.

```
<launch>

  <group ns="namespace">
    <node pkg="Package name" name="node name given"
          type="original node name"/>
  </group>

  (or)

  <node pkg="Package name" name="node name given"
        type="original node name"/>

  ⋮ ⋮ ⋮

</launch>
```

ROS Tutorial {Intermediate Level}

① Roslaunch tips for large projects

⇒ Large applications on a robot typically involve several interconnected nodes, each of which have many parameters.

② Running ROS across multiple machines

⇒ ROS is designed with distributed computing in mind.

↳ Driver node that communicates with a piece of hardware must run on the machine to which the hardware is physically connected.

CHAPTER 3

ETH zuric cource

ETH zürich (ROS Course)

① Lecture 1

ROS = Robot Operating System

→ It is not an Operating System

It is middle where that sits between operating system and actual program that you write.

Plumbing	Tools	Capabilities	Ecosystem
→ Process management	→ Simulation	→ Control	→ Package Organization
→ Inter-Process communication	→ Visulization	→ Planning	→ Software distribution
→ Device drivers	→ Graphical user Interface.	→ Perception	→ Documentation
	→ Data logging	→ Mapping	→ Tutorials
		→ Manipulation	

★ Ros Philosphy

Peer to Peer

⇒ Individual programs communicates over defined API (ROS messages, Services etc..)

Distributed

→ Programs can be run on multiple computers and communicate over the network.

Multi-lingual

→ ROS modules can be written in any language for which a client library exists (C++, Python, Java etc).

Light-Weight

Free and Open-Source

★ ROS Master

→ Manages the communication between nodes.

→ Every node registers at startup with the master.

`roscore` ⇒ To start ROS Master

★ ROS Node

→ Single-purpose

→ Executable Program

→ Individually compiled, executed and managed.

→ Organized in package.

`rosrun` package_name node_name
↳ To run the node

`rosnode list` ⟶ To list all the active nodes

`rosnode info` node_name → Retrieve information about a node.

* ROS Topic

⇒ Nodes communicate over topics
- → Nodes can publish or subscribe to a topic
- → Typically 1 publisher and n subscribers

⇒ Topic is a name for a stream of messages.

`rostopic list` ⇒ List active topics

`rostopic echo` /topic ⇒ Subscribe & print the Contents of a topic.

`rostopic info` /topic ⇒ Shows information about the topic

* ROS Messages

⇒ Data structure defining the type of a topic.

⇒ Defined in *.msg file

`rostopic type` /topic
→ To See the type of a topic.

`rostopic pub` /topic type args
→ Publish message to a topic

⇒ Ros Messages can be nested

`rostopic hz` /topic
→ Frequency of Publish in Hz

Catkin workspace

- src → The source space contains the source code.
- build → where CMake is invoked to build the package in the source space.
  → cache information and other intermediate files are kept here.
- devel → The development (devel space is where built targets are placed

Catkin clean ⇒ Cleans the entire build and devel space without affecting the src folder

Catkin config → Catkin workspace setup can be checked here.

ROS Launch

⇒ launch is a tool for launching multiple nodes (as well as setting parameters)

⇒ Are written in XML as *.launch file.

⇒ If not yet running, launch automatically starts a roscore.

roslaunch file_name.launch
→ {If you are already in the folder}

roslaunch Package_name file_name.launch
→ {Start launch file from a package}

```
<launch>
    <node name="Given name" pkg="Package name" type="original node name"
          output="screen"/>
<launch>
```

⇒ launch file can be nested.

★ <arg> tag

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

$(arg arg-name) → Can be used in launch file

⇒ Launching launch file with argument

roslaunch Launch_file.launch arg_name:=Value

⇒ Including other launch file

```
<include file="$(find Package_name)/relative path.launch"/>
```

* Gazebo Simulator

→ Simulate 3d rigid-body dynamics
→ Simulate a variety of sensors including noise.
→ 3d visualization and user interaction.
→ Include a database of many robots and environments
→ Provides a ROS interface
→ Extensible with plugins

⇒ To run Gazebo with ROS

roscore run gazebo_ros gazebo

① Lecture 2

★ ROS Package

⇒ ROS software is organized into packages, which can contain source code, launch files, configuration files, messages, definations, data and documentation.

⇒ A package that builds up on/requires other packages, declares these as dependencies.

⇒ To Create new package

`Catkin_Create_PKg` Packagename {dependencies}

→ In Src folder

⇒ Separate the message definition Package from other package.

★ Package.xml

⇒ The package.xml file defines the properties of the package.

- Package name
- Version number
- Authors
- Dependencies on other pakage
- ...

* CMakeList.txt

↳ Input to CMake build System.

1. Requires CMake version

cmake_minimum_required

2. Package Name

project()

3. Find other CMake/Catkin packages needed for build.

find_package()

4. Message/Service/Action Generators

add_message_files()
add_service_files()
add_action_files()

5. Invoke message/service/action generation

generate_messages()

6. Specify package build info export.

catkin_package()

7. Libraries/Executables to build

add_library()
add_executable()
target_link_libraries()

8. Tests to build

catkin_add_gtest()

9. Install rules

install()

★ ROS C++ Client Library (roscpp)

```
#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;
}
```

★ Node Handle

⇒ There are four main types of Node handles :-

1. Default (Public) node handle :

```
nh_ = ros:: NodeHandle();
```

2. Private node handle:

```
nh_private_ = ros::NodeHandle("~");
```

3. Namespaced node handle:

```
nh_eth_ = ros::NodeHandle("eth");
```

* Logging

⇒ Mechanism for logging human readable text from nodes in the Console and to log file.

⇒ Automatic logging to Console, log file, and /rosout topic.

⇒ Different Severity levels:

Debug → Info → Warn → Error → Fatal.

⇒ Supports both printf and stream-style formatting.

```
ROS_INFO("Result: %d", result);
ROS_INFO_STREAM("Result: " << result);
```

⇒ Further features such as Conditional, throttled delayed logging etc..

:-

* ROS Parameter Server

⇒ Nodes use the Parameter server to Store and retrive parameters at runtime.

⇒ Best used for Static data such as Configuration Parameter

⇒ Parameters can be defined in launch files or Seperate YAML file.

{Example

Config.yaml

```
Camera:
  left:
    name: left_Camera
    exposure: 1
  right:
    name: right_Camera
    exposure: 1.1
```

{launching it through launch file}

```
<node name="Name" PKg="Package" type="node.type">
    <rosparam command="load"
        file="$(find package)/config/config.yaml"/>
</node>
```

C++ API for Parameter Server

⇒ To get a parameter in C++:

NodeHandle.getParam(Parameter_name, Variable)

★ 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 with

rosrun rviz rviz

topics

xaml

③ Lecture 3

★ TF Transformation System

⇒ Tool for Keeping track of coordinate frames over time.

⇒ Maintains relationship between coordinate frames in a tree structure buffered in time.

⇒ Lets the user transform point, vectors, etc. between coordinate frames at desired time.

⇒ Implemented as publisher/subscriber model on the topic /tf and /tf_static

Tools

(A) Command line

`rosrun tf tf_monitor`

↳ Prints information about the current transform tree.

`rosrun tf tf_echo` source_frame target_frame

↳ Prints information about the transformation between two frame.

(B) View Frames

```
rosrun tf view_frames
```

↳ Creates a visual graph (PDF) of the transform tree.

(C) RViz

3D Visulization of the transformation

* RQT User Interface

- ↳ Custom interface can be setup
- ↳ Lots of existing plugins exists
- ↳ Simple to write own plugins

⇒ Run RQT with

```
rosrun rqt_gui rqt_gui
```

```
# rqt_image_view
# rqt_multiplot
# rqt_graph
# rqt_Consol
# rqt_logger_level
```

* Robot Models

(Unified Robot Description Format)
(URDF)

⇒ Defines an XML format for representing a robot model.

⇒ Kinematic & dynamic description

⇒ Visual representation

⇒ Collision model

⇒ URDF generation can be scripted with XACRO.

link Joint

Example : ~~robot.urdf~~ link

```
<link name="link_name">
  <visual>
    <geometry>
      <mesh filename="mesh.dae"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
```

```
<inertial>
  <mass value="10" />
  <inertia ixx="0.4" ixy="0.0" ... />
</inertial>
</link>
```

Example: Joint

```
<joint name="joint_name" type="revolute">
  <axis xyz="0 0 1" />
  <limit effort="1000.0" upper="0.548" .../>
  <origin rpy="0 0 0" xyz="0.2 0.01 0" />
  <parent link="parent_link_name"/>
  <child link="child_link_name"/>
</joint>
```

⇒ The robot description (URDF) is stored on the parameter server (typically) under /robot description.

⇒ You can visualize the robot model in Rviz with the RobotModel plugin.

★ Simulation Description

(Simulation Description Format)
(SDF)

→ Standard format for Gazebo

⇒ Defines an XML format to describe

- # Environment (lighting, gravity etc)
- # Objects (static & dynamic)
- # Sensors
- # Robot.

⇒ Gazebo Converts a URDF to SDF automatically.

---

ros::Duration

```
ros::Duration duration(0.5);
```

ros::Rate

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

★ ROS Bags (*.bag)

⇒ Bag is a format used to store message data

⇒ Suited for logging and recording dataset for later visualization and analysis.

④ Lecture 4

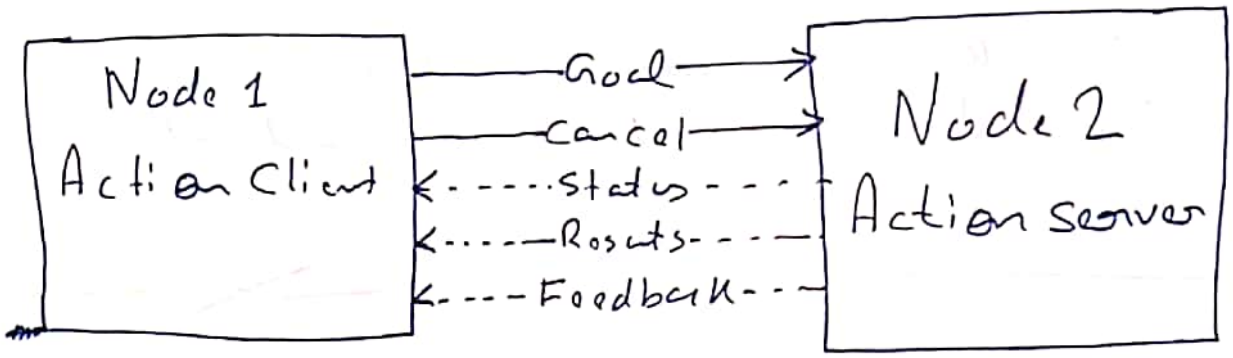
★ ROS Services

⇒ Request/response communication between nodes is realized with services.

⇒ Services are defined in *.srv file.

★ ROS Actions (actionlib)

↳ Very similar to services but meant for requests that take a longer time.



*.action file

★ ROS time

↳ Normally, ROS uses the PC's system clock as time source (wall time)

⇒ To take advantage of the simulated time, you should always use the ROS Time APIs:

⊕ ros::Time

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

* Debugging Strategies

⇒ Compile and run code often to catch bugs early.

⇒ Understand compilation and runtime error messages.

⇒ Use analysis tools to check data flow.

⇒ Visualize and plot data

⇒ Divide program into smaller steps and check intermediate results.

⇒ Make your code robust with argument and return value checks and catch exceptions

CHAPTER 4

TF Turorial

TF Tutorial

① Introduction to TF

`rosrun tf view_frames`

→ Creates a diagram of the frames being broadcast by tf over ROS.

`rosrun rqt_tf_tree rqt_tf_tree`

`rosrun tf tf_echo [reference_frame] [target_frame]`

→ Reports the transformation between any two frame broadcast over ROS.

⇒ Translation

Rotation in Quaternion

in Roll Pitch Yaw

① Writing a tf broadcaster (C++)

`#include <tf/transform_broadcaster.h>`

⇒ The tf package provides an implementation of a TransformBroadcaster to help make the task of publishing transforms easier.

⇒ To use the TransformBroadcaster, we need to include the tf/transform_broadcaster.h

br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", turtle_name));

Name of the parent frame

Name of the child frame

② Writing a tf listener (C++)

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

→ The tf package provides an implementation of a TransformListener to help make the task of receiving transforms easier.

→ To use the TransformListener, we need to include the tf/transform_listener.h header file.

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

→ Here we create TransformListener object.

→ Once the listener is created, it starts receiving tf transformations over the wire, and buffers them for up to 10 sec.

```
try {
    listener.lookupTransform("/turtle2", "/turtle1",
                             ros::Time(0), transform);
}
```

→ We want the transform from frame /turtle1 to /turtle2.

→ The time at which we want to transform. Providing ros::Time(0) will get us the latest available transform.

→ The object in which we want to store the resulting transform.

③ Adding a frame (C++)

* Why adding frames

For many tasks it is easier to think inside a local frame.

↓

eg: It is easier to reason about a laser scan in a frame at the center of the laser scanner.

* Where to add frames

⇒ tf builds up a tree structure of frames. it does not allow a closed loop in the frame structure.

↳ Frame only has one single parent, but it can have multiple childrens.

* How to add a frame

```
transform.setOrigin(tf::Vector3(0.0, 2.0, 0.0));
transform.setOrigin(tf::Quaternion(0,0,0,1));
br.sendTransform(tf::StampedTransform
(transform, ros::Time::now(), "turtle1", "carrot1"));
```

Parent ⇒ turtle1

Child ⇒ carrot1

(4) Learning about tf and time (C++)

* tf and Time

⇒ tf keeps track of a tree of coordinate frames.

↳ This tree changes over time, and tf stores a time snapshot for every transform (for up to 10 sec by default)

*Wait for transform

```
listener.waitForTransform("/turtle2", "/turtle1", now
                          , ros::Duration(3.0));
```

waitForTransform() will actually block until the transform between the two turtle becom available until the timeout has been reached.

(5) Time travel with tf (c++)

```
try{
  ros::Time now = ros::Time::now();
  ros::Time past = now - ros::Duration(5.0);
  listener.waitForTransform("/turtle2", now,
                            "/turtle1", past
                            ,"/world", ros::Duration(1.0));

  listener.lookupTransform("/turtle2", now,
                           "/turtle1", past,
                           "/world", transform)
```

CHAPTER 5

URDF Tutorial

Urdf Tutorial

① Building a Visual robot model with URDF from scratch

```
<?xml version="1.0"?>
<robot name="Name of Robot">
```

{ Material definition }

{ Link 1 definition }

{ joint 1 definition }

{ Link 2 definition }

{ joint 2 definition }

⋮ ⋮ ⋮

```
</robot>
```

★ Material definition

```
<material name="Name of material">
    <color rgba="0 0 0.8 1"/>
</material>
```

* Link defination

```
<link name="Name of the link">
  <visual>
    <geometry>
      (geometry defination)
    </geometry>
    <origin rpy="0.0 0" xyz "0 0 0"/>
    <material name="Name of material to apply" />
  </visual>
</link>
```

Controls position and orientation of mesh relative to origin

⊕ Geometry defination

```
<cylinder length="0.6" radius="0.2" />
<box size="0.6 0.1 0.2" />
<sphere radius="0.2" />
```

For loading the basic shapes

```
<mesh filename="Package://urdf_tutorial/meshes
                /1-finger.dae" />
```

For loading the mesh file

* Joint defination

```
<joint name="name_of_joint" type="fixed">
    <Parent link="parent link name"/>
    <Child link="Child link name"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
```

Controls position and orientation of child link wrt Parent link

stl → Only mesh data.

dae → mesh + ~~color~~ material data

② Building a Movable Robot model with URDF

Joint types ⇒ Continuous, Revolute and Prismatic.

- Continuous: Can take any angle from $-\infty$ to $+\infty$
  - `<axis xyz="0 0 1" />` → Axis of rotation
- Revolute: They rotate in the same way that the continuous joints do, but they have strict limits
  - `<axis xyz="0 0 1" />`
  - `<limit effort="1000" lower="0.0" upper="0.548" velocity="0.5" />`
    - To limit effort (torque), rotation, angular velocity

Prismatic

`<limit effort="1000" lower="-0.38" upper="0" velocity="0.5" />`

★ Other types of joint

- Planar joint
- Floating joint

③ Adding physical and Collision properties to a Urdf model

* Collision

⇒ Two main reason to make different visual and collision model :-

① Quicker processing

② Safe zone around the robot.

* Physical properties

① Inertia

→ Kg

```
<inertial>
    <mass value="10" />
    <inertia ixx="0.4" ixy="0.0" iyy="0.4"
             iyy="0.4" iyz="0.0" izz="0.2" />
</inertial>
```

⇒ You can also specify an origin tag to specify the center of gravity and the inertial reference frame (relative to the link's reference frame).

② Contact coefficients

⇒ This is done with a subelement of the collision tag called contact_coefficients.

mu ⇒ friction coefficient
Ke ⇒ Stiffness coefficient
Kd ⇒ Dampening coefficient.

③ Joint Dynamics

⇒ How the joint moves is defined by the dynamics tag for the joint.

↳ There are two attributes here :-

friction ⇒ Static friction

damping ⇒ physical damping value.

(9) Using xacro to clean up a URDF file

⇒ xacro package does three things that are very helpful.

(i) Constants
(ii) Simple math
(iii) Macros

⇒ xacro is a macro language.

* **Constants** {Name given to the Name Space}

```
<xacro:property name="width" value="0.2" />
```

{defining the variable width with value 0.2}

⇒ Using that value → "${width}"

* **Simple math**

"${5/6}" ⟹ "0.8333..."

* Macros

Simple macro

```
<xacro:macro name="default_origin">
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
```

↓ {using it}

```
<xacro:default_origin/>
```

Parameterized macro

```
<xacro:macro name="default_inertial"
             params="mass">
    <inertial>
        <mass value="${mass}" />
        <inertia .... />
    </inertial>
</xacro:macro>
```

↓ {using it}

```
<xacro:default_inertial mass="10" />
```

(5) Using a URDF in Gazebo

(1) Gazebo plugin

⇒ To get ROS to interact with Gazebo, we have to dynamically link to the ROS library that will tell Gazebo what to do.

⇒ To link Gazebo and ROS, we specify the plugin in the URDF, right before closing </robot> tag.

```
<gazebo>
  <plugin name="gazebo_ros_control"
          filename="libgazebo_ros_control.so">
    <robotNamespace></robotNamespace>
  </plugin>
</gazebo>
```

(2) Transmission

⇒ For every non fixed joint, we need to specify a transmission, which tells Gazebo what to do with the joint.

Example

```
<transmission name="head_swivel_trans">
    <type> transmission_interface/SimpleTransmission
                                        </type>
    <actuator name="$head_swivel_motor">
        <mechanicalReduction> 1 </mechanicalReduction>
    </actuator>
    <joint name="head_swivel">
        <hardwareInterface>PositionJointInterface
                    </hardwareInterface>
    </joint>
</transmission>
```

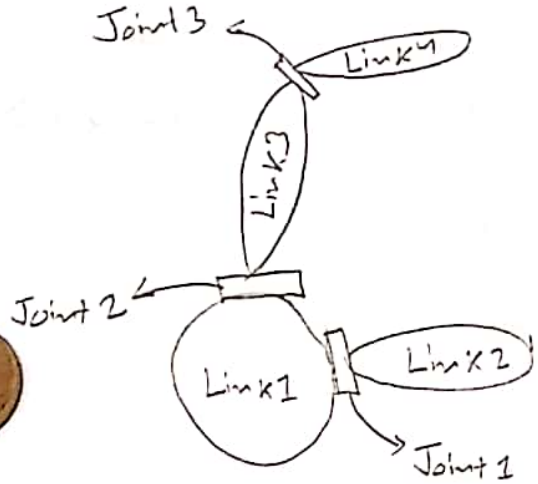
⇒ Just treade most of this chunk of code as boilerplate.

{ refers to section of code that have to be included in many places with little or no attention }

{ This should match the joint name }

Learning URDF (including C++ API)

1. Create your own urdf file



Parsing

It is the process of analyzing text made of a sequence of tokens to determine its grammatical structure with respect to a given formal grammar.

⇒ There is a simple command line tool that will parse a urdf file for you, and tell you if the syntax is correct.

↳ liburdfdom-tool

```
sudo apt-get install liburdfdom-tools
```

⇒ Using the tool:

```
check_urdf my_robot.urdf
```

⇒ You can visulize the URDF using graphiz.

`urdf_to_graphiz myrobot.urdf`

→ Generates Pdf

2. Parse a urdf file

⇒ Convention of Storing robot in package name:

MYROBOT_description

- → Package.xml
- → CMakeLists.txt
- → /urdf
- → /meshes
- → /materials
- → /cad.

⇒ You can use urdf package to parse your urdf file

`urdf::Model model;`

`model.initFile(urdf_file)`

- True if successfully Parsed
- False if not

3. Using the robot State Publisher on your own robot

⇒ When you are working with a robot that has many relevant frames, it becomes quite a task to publish them all to tf.

↳ The robot State publisher is a tool that will do this job for you.

↓

The robot State publisher helps you to broadcast the State of your robot to the tf transform library.

(a) Running as a node

{Easiest way to run the robot state publisher is as a node.}

⇒ You need two things to run the robot state Publisher:

(i) Urdf xml robot description loaded on the parameter server.

(ii) A source that publishes the joint positions as a Sensor_msgs/JointState.

⑤ Running as a library

⇒ Advanced users can also run the robot state publisher as a library, from within their own C++ code.

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

RobotStatePublisher(const KDL::Tree& tree);
```

↓

{Constructor which takes KDL tree}

⇒ Now every time you want to publish the state of your robot, you call the publishTransforms function:

```
void publishTransforms(const std::map<std::string, double>& joint_positions, const ros::Time& time);
```

4. Start using the KDL parser

(a) Building the KDL parser

```
rosdep install kdl_parser
rosmake kdl_parser
```

(b) Using in your code

⇒ First add the KDL parser as a dependency to your package.xml.

```
<build_depend package="kdl_parser"/>
<run_depend package="kdl_parser"/>
```

⇒ To start using the KDL parser in your C++ code, include the following file:

```
#include <kdl_parser/kdl_parser.hpp>
```

⇒ Now there are different ways to proceed. You can construct a KDL tree from a urdf in various forms.

- From a file
- From a parameter server
- From an xml element
- From a urdf model

5. Using urdf with robot_state_publisher

- First, we create the URDF model with all the necessary parts.
- Then we write a node which publishes the JointState and transforms.
- Finally we run all the parts together.

Robot model

⇒ All the robot model related files are kept in a package with name robotname_description

↳ General convention

⇒ General directory tree of robotname_description:

robotname_description

- → CMakeLists.txt { The usual stuff }
- → package.xml
- → meshes { This directory is further subdivided into different directories according to different sub assemblies or sub categories, that contains mesh files (Stl, dae) }
  - Stl → { only mesh data (binary) }
  - dae → { mesh + color data }

Example

- → base
- → head
- → gripper
- → Sensor

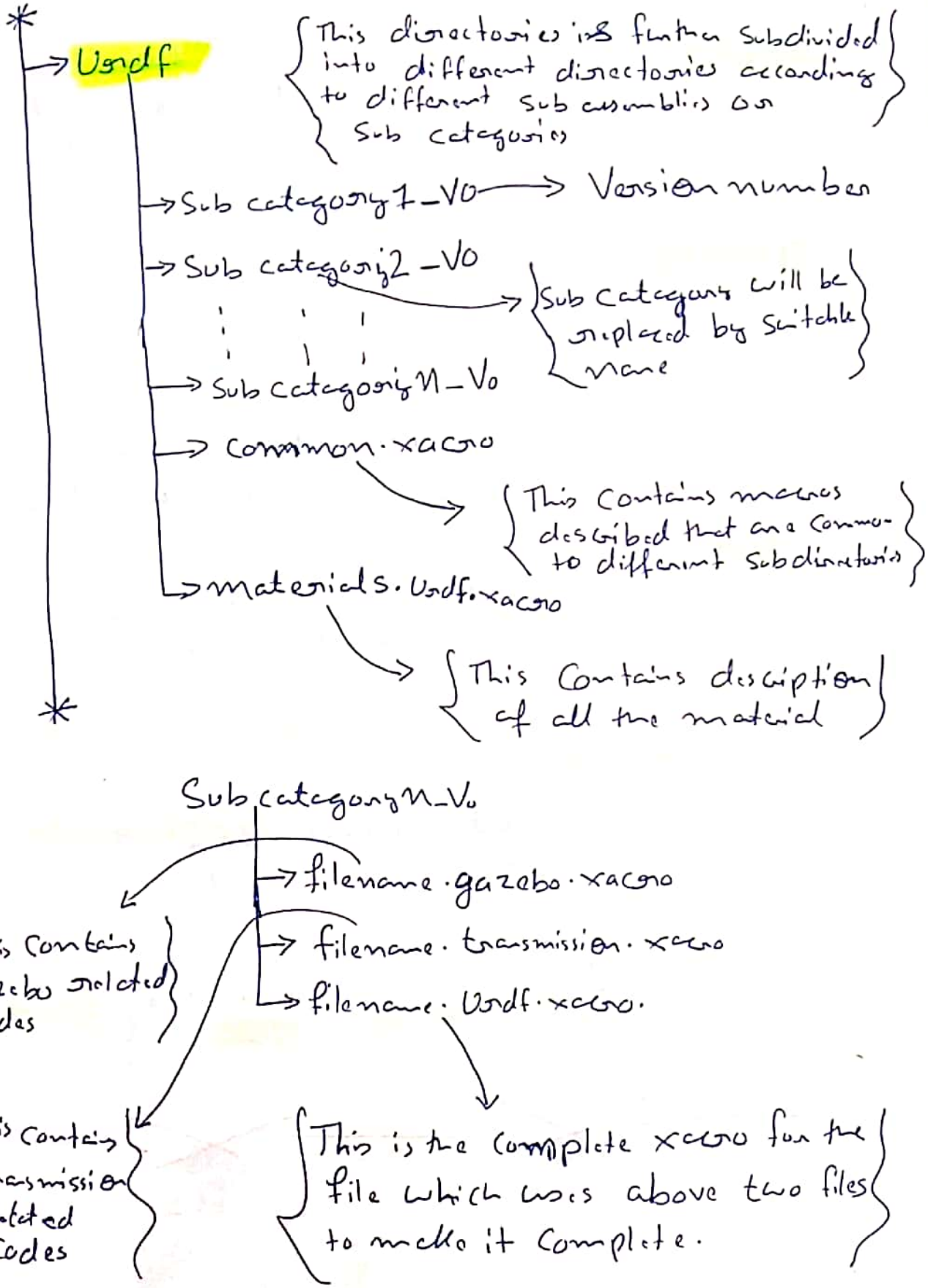
- → materials
  - → texture { Contains textures that may be applied }

*

{ Thi gaz co

{ Th to

Scanned by CamScanner



*

→ **robots**

This directory uses all the xacro defined in urdf directory to create different assemblies of robots, with different features.

→ **gazebo**

→ gazebo.urdf.xacro (directory)

This file is used by files in robot to add gazebo Plugin and add basic settings

→ **test**

This contains test files to ensure all the files work together Perfectly.

→ **documents**

This generally contains .xls files, diagrams etc.

For all Engineering drawings

for all the Values

CHAPTER 6

Mastering ROS for Robotics Program

Chapter -2

Mastering_ROS_for_Robotics_Program

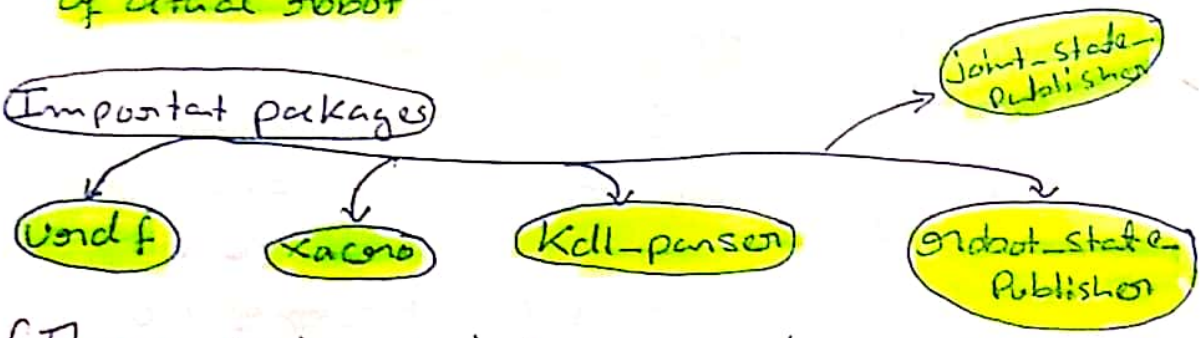
by → Lentin_Joseph

⇒ The first phase in robot manufacturing is its design and modeling.

→ { Blender } { Solid works }

→ { One of the main purposes of modeling is simulation }

" The Virtual robot model must have all the characteristics of real hardware, the shape of robot may or may not look like the actual robot but it must be an abstract, which has all the physical characteristics of actual robot "



{ These packages help us Create the 3D robot model description with the exact characteristics of real hardware.

→ We can define a robot model, sensors, and a working environment using URDF.

→ Flexible links can't be represented using URDF.

* joint_state_publisher

⇒ This package contains a node called joint_state_publisher, which reads the robot model description, find all joints, and publishes joint value to all non fixed joints using GUI slider.

* Kdl_parser

⇒ Kinematic and Dynamic library (KDL) is an ROS package that contains parser tools to build a KDL tree from the URDF representation.

↳ used to publish the joint states and also to forward and inverse kinematics of the robot.

* robot_state_publisher

⇒ This package reads the current robot joint states and publishes the 3D poses of each robot link using the kinematics tree build from the urdf.

↳ 3D pose of the robot is published as ROS tf.

↳ Publishes the relationship between coordinates frames of a robot.

gazebo tag in urdf

→ Used when we include the simulation Parameters of the gazebo simulator inside URDF.

→ We can use this tag to include gazebo plugins, gazebo material properties etc.

⇒ The collision and inertia parameters are required in each link; otherwise, Gazebo will not load the robot model properly.

Conversion of xacro to URDF

```
rosrun xacro xacro.py filename.xacro > filename.urdf
```

⇒ We can use the following line of code in the ROS launch file for converting xacro to URDF and use it as a robot_description parameter:

```
<param name="robot_description" command="$(find xacro)/xacro.py $(find packagename)/urdf/filename.xacro" />
```

* transmission tag

⇒ Transmission tag relates a joint to an actuator.

⇒ It defines the type of transmission that we are using in a particular joint and type of motor and its parameters.

⇒ It also defines the type of hardware interface we use when we interface with the ROS controllers.

* Understanding joint state publisher

⇒ It finds the **non fixed joints** from the URDF model and publishes the joint state values of each joint in the **sensor_msg/JointState** message format.

⇒ If we set **use_gui to true**, the joint_state_Publisher node displays a slider based Control window to control each joint.

↳ The lower and upper value of a joint will be taken from the lower & upper values associated with the limit tag used inside the joint tag.

* Understanding the robot state publisher

⇒ Robot state publisher package helps to **Publish the state of the robot to tf.**

⇒ This package **subscribes to joint states** of the robot and **publishes the 3D pose** of each link using the kinematic representation from the URDF model

CHAPTER 7

Simulating Robot Using ROS and Gazebo

Chapter - 3

Simulating Robot Using ROS and Gazebo

⇒ Robot simulation will give you an idea about the working of robots in a virtual environment.

Gazebo: Multirobot simulator for complex indoor and outdoor robotic simulation.

★ **Some important packages**

- **gazebo_ros_pkgs**
  - → This contains wrappers and tools for interfacing ROS with Gazebo.
- **gazebo_msgs**
  - → This contains messages and service data structures for interfacing with Gazebo from ROS.
- **gazebo_plugins**
  - → This contains Gazebo plugins for sensors, actuators and so on.
- **gazebe_ros_control**
  - → This contains standard controllers to communicate between ROS and Gazebo.

★ Converting to Gazebo

Required

⇒ An <inertial> element within each <link> element must be properly specified and configured.

Optional

⊙ Add <gazebo> element for every <link>
- → Convert visual colors to Gazebo format.
- → add Sensor plugins

⊙ Add <gazebo> element for every <joint>
- → Set proper damping dynamics
- → Add actuator control plugins.

⊙ Add a <gazebo> element for the <robot> element

⊙ Add <link name="world"/> link if the robot should be rigidly attached to the world.

★ The <gazebo> Element

⇒ The <gazebo> element is an extension to the URDF used for specifying additional Properties needed for simulation purposes in Gazebo.

⇒ There are three different of <gazebo> elements:-
- → One for <robot> tag
- → One for <link> tag
- → One for <joint> tag

⇒ If a <gazebo> element is used without a reference Property, it is assumed the <gazebo> element is for the whole robot model.

★ Material in gazebo

```
<gazebo reference="link1">
    <material> Gazebo/Orange </material>
</gazebo>
```

★ <gazebo> Elements for link

- → Material
- → gravity (bool)
- → damping factor
- → maxVel
- → min Depth
- → mu1
- → mu2
- → fdir1
- → Kp
- → Kd
- → self Collide
- → max Contacts
- → lesen Ratio

★ <gazebo> Elements for Joints

- → StopCfm
- → StopErp
- → provideFeedback
- → implicitSpringDamper
- → CfmDamping
- → fudgeFactor

⇒ To Verify if your URDF can be properly converted into a SDF.

```
gz sdf -p MODEL.urdf
```

---

* Adding transmission tag to actuate the model

⇒ In order to actuate the robot using ROS controllers, we should define the <transmission> element to link actuators to joint.

```
<transmission name="tran1">
    <type> transmission_interface/SimpleTransmission
    </type>
    <joint name="joint_name">
        <hardwareInterface> PositionJointInterface
        </hardwareInterface>
    </joint>
    <actuator name="motor1">
        <mechanicalReduction> 1 </mechanicalReduction>
    </actuator>
</transmission>
```

⇒ Transmission <type>

↳ Currently, transmission_interface/SimpleTransmission is only supported.

⇒ The <hardwareInterface> element is the type of hardware interface to load (Position, Velocity, or effort interface)

↓

{hardware interface is loaded by the gazebo_ros_control plugin}

* Adding the gazebo_ros_control plugin

↳ In order to parse the transmission tags and assign appropriate hardware interface and the control manager.

```
<gazebo>
  <Plugin name="gazebo_ros_control"
          filename="libgazebo_ros_control.so">
    <robotNamespace> / </robotNamespace>
  </Plugin>
</gazebo>
```

⇒ If we are not specifying the name, it will automatically load the name of the robot from the URDF.

⇒ We can also specify:

- Controller update rate — <controlPeriod>
- location of robot description on Parameter Server — <robotParam>
- Type of robot hardware interface — <robotSimType>
  - → JointStateInterface
  - → EffortJointInterface
  - → VelocityJointInterface

* Adding Sensor in Gazebo

⇒ To build a sensor in Gazebo, we have to model the behavior of that sensor in Gazebo.

↳ There are some probuilt sensor models in Gazebo that can be used directly in our code without writing a new model.

```
<robot>
  ... robot description ...
  <link name="sensor_link">
    ---- Link description ---
  </link>

  <gazebo reference="sensor_link">
    <sensor type="sensor type" name="name of sensor">
      ... Sensor Parameters ...
      <plugin name="name of plugin" filename="libgazebo_ros
        _name.so">
        ... Plugin parameters ...
      </plugin>
    </sensor>
  </gazebo>

</robot>
```

* <Sensor> tag for laser Scanner

<Sensor type="ray" name="name of sensor">

<Pose> 0 0 0 0 0 0 </Pose>

<Visualize> false </Visualize>

<update_rate> 40 </update_rate>

{⇒ frequency at which it gives new data}

{When true a semi-translucent laser ray is visualized within the scanning zone}

{Position and Orientation of Sensor}

<ray>

<scan>

<horizontal>

<samples> 720 </samples>

<min_angle> -1.57 </min_angle>

<max_angle> 1.57 </max_angle>

</horizontal>

</scan>

<range>

<min> 0.10 </min>

<max> 10.0 </max>

<resolution> 0.001 </resolution>

</range>

</ray>

```
<plugin name="Name of Controller"
        filename="libgazebo_ros_laser.so">
    <topicName> /scan </topicName>
    <frameName> Name of link </frameName>
</plugin>

</sensor>
```

★ Ros controllers

⇒ On each joint, we need to attach a Controller that is compatible with the hardware interface mentioned inside the transmission tag.

⇒ ROS Controller mainly consists of a feedback mechanism, most probably a PID loop, which can receive a set point, and control the output using the feedback from the actuator.

⇒ The ROS controllers are provided by a set of packages called ros_control.

⇒ The ros_control packages are composed of the following individual packages:

- control_toolbox
  - Contain common modules (PID & Sine) that can be used by all controllers.
- Controller_interface
  - Contains the interface base class for controllers.
- Controller_manager
  - Infrastructure to load, unload, start and stop controllers.
- Controller_manager_msgs
  - This provides the message and service definition for the Controller manager.
- hardware_interface
  - Contains base class for the hardware interface.
- transmission_interface
  - Contain the interface classes for the transmission interface.
    (differential, four bar linkage, joint state, Position and Velocity)

* Different type of ROS Controllers and Hardware Interface

Joint_position_Controller
Joint_State_Controller
Joint_effort_Controller

Joint Command Interface
  # Effort Joint Interface
  # Velocity Joint Interface
  # Position Joint Interface
Joint State Interface

CHAPTER 8

ROS Controllers

Ros Controllers

⇒ The ros_control package provide a set of controller plugins in order to interact in different ways with the joints of your robot.

1. effort_controllers
   - ↳ joint you want to control accept effort commands.
   - Joint_effort_controller
     - ↳ accepts effort as input value
   - joint_position_controller
     - ↳ accepts position as input value
   - joint_velocity_controller
     - ↳ accepts velocity as input.
2. Position_controllers
   - ↳ joint you want to control accepts position commands.
   - joint_position_controller
3. Velocity_controllers
   - ↳ joint you want to control accepts Velocity commands.
   - joint_velocity controller
4. joint_state_controller
   - ↳ This plugin provides the state of the joints, Publishing them into a topic called /joint_states.
   - joint_state_controller

* Configration file

robotname_control.yaml

```
robotname:
    joint_State_Controller:
        type: joint-State_Controller / JointStateController
        Publish_rate: 50

    joint1_position_controller:
        type: effort_Controllers/ JointPositionController
        Joint: Jointname
        Pid: {P:100.0, i:0.01, d: 10.0}
```

* Launch file

```
<launch>
    <include file="$(find my_package)/launch/Robotname
                _world.launch"/>

    <rosparam file="$(find my-package)/config /
                Robotname_gazebo_control.yaml"
                command="load"/>

    <node name="Controller_Spawner" PKG="controllermanager"
        type="Spawner" respawn="false" output="screen"
        ns="/Robotname  args=" joint_State Controller
                joint1_Position_Controller
                :  :  :  :
                />
```

<node name = "robot_state_publisher" pkg = "robot_state_publisher"
type = "robot_state_publisher"
respawn = "false" output = "screen">
<remap from="/joint_states" to="/robotname/joint_states" />

</node>

</launch>

★ Transmission tag

→ Attribute name (required).

→ Elements

→ <type> {Specifies the transmission type}

→ <joint>

→ Attribute name
{Joint the transmission is connected to}

→ <hardwareInterface>
{Specify a supported joint-space hardware interface}

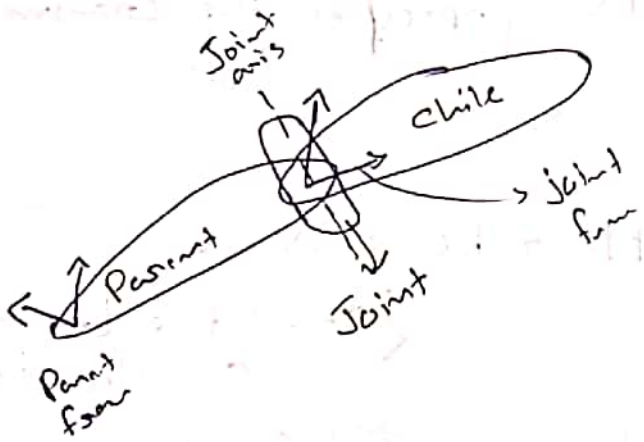
→ <actuator>
{The actuator the transmission is connected to}

*

* → Attribute name {name of the actuator}
→ <mechanical Reduction> {Specifies a mechanical reduction at the joint/actuator transmission}

★ Joint tag

⇒ The joint describe the Kinematics and dynamics of the joint and also specifies the Safety limits of the joint.



Attributes

→ name {Unique name for the joint}

→ type {Specify the type of joint}

*

* → revolute { hinge joint with limited range.
→ continuous { hinge joint with no limit on range
→ prismatic { Sliding joint that slides along the axis, and has a limited range
→ fixed { This is not really a joint because it cannot move
→ floating { Have all 6 DOF }
→ planar { Allows motion in a plane ⊥ to the axis }

Elements

→ <origin>
  { This is the transform from the Parent link to the child link }
  → xyz { Represents xyz offset }
  → rpy { Represents rotation about the corresponding axis. }

→ <parent>
  → link { Parent link name }

→ <child>
  → link { child link name }

* → <axis> → xyz

→ <calibration>
{The reference positions of the joint, used to calibrate the absolute position of the joint}

→ rising
{when the joint moves in a positive direction, this reference position will trigger a rising edge}

→ falling
{When the joint move in a positive direction, this reference position will trigger a falling edge}

→ <dynamic>
{for specifying physical properties of joint

→ damping

→ friction

→ <limit> (required for revolute and prismatic joint

→ lower

→ upper

→ effort {for enforcing max joint effort}

→ velocity {for enforcing max joint velocity}

→ <mimic>

→ Attribute joint (to mimic)

→ multiplier

→ offset

⇒ This tag is used to specify that the defined joint mimics another existing joint.

⇒ The value of the joint can be computed as

$$\text{Value} = (\text{multiplier} \times \text{other\_joint\_value}) + \text{offset}$$

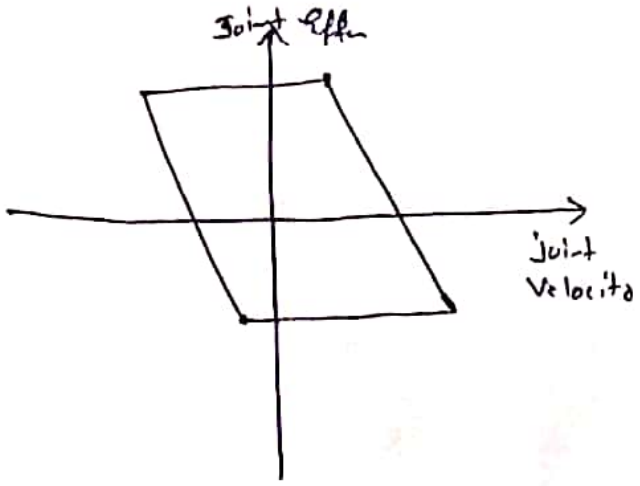
→ <Safety_Controller>

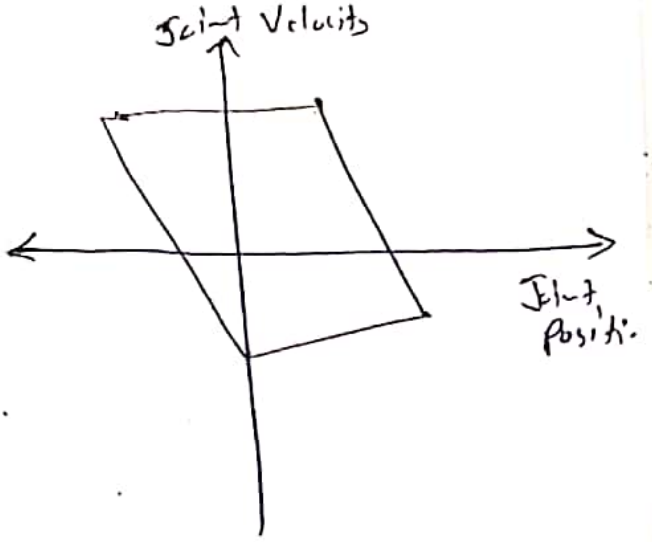
→ soft_lower_limit

→ soft_upper_limit

→ k_position

→ k_velocity





CHAPTER 9

Navigation Stack

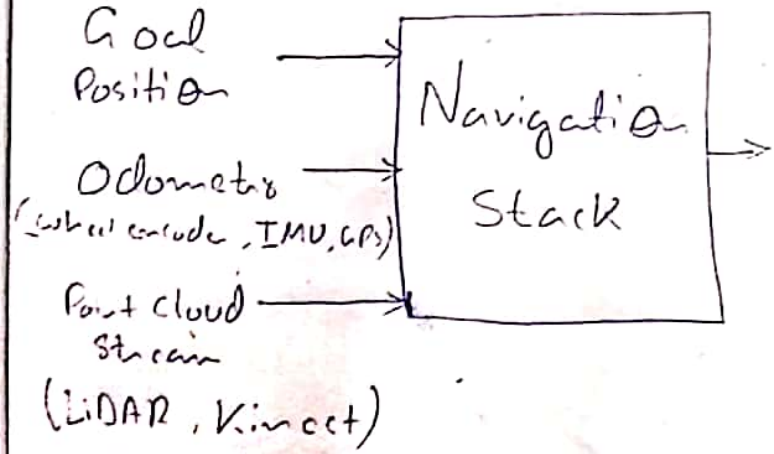
Navigation Stack

⇒ Mobile robot navigation.

- Simultaneous Localization and Mapping (SLAM)
- Adaptive Monte Carlo Localizatio (AMCL)

* Understanding ROS Navigation stack

⇒ The main aim of the ROS navigation package is to move a robot from the start position to the goal position, without making any collision with the environment.



Inputs to Navigation Stack

⊙ Odometry Source

⇒ Odometry data of a robot gives the robot position with respect to its starting position.

⇒ Main odometry source(s) :-

- → Wheel encoders
- → IMU
- → 2D/3D Cameras. (Visual odometry)

⇒ The Odom value should publish to the Navigation Stack, which has a message type of nav_msgs/Odometry.

↓

{Can hold position & Velocity of robot}

⊙ Sensor Source

⇒ We have to provide laser scan data or point cloud data to the Navigation stack for mapping the robot environment.

⇒ This data, along with odometry, combines to build global and local Costmap.

⇒ The data should be of type Sensor_msgs/LaserScan or sensor_msgs/PointCloud

⊙ Sensor transforms /tf

⇒ The robot should publish the relationship between the robot coordinate frame using ROS tf.

⊙ Base_Controller

⇒ The main function of the base Controller is to convert the output of the Navigation Stack, which is twist (geometry_msgs/Twist) message, and convert it into corresponding motor velocities of the robot.

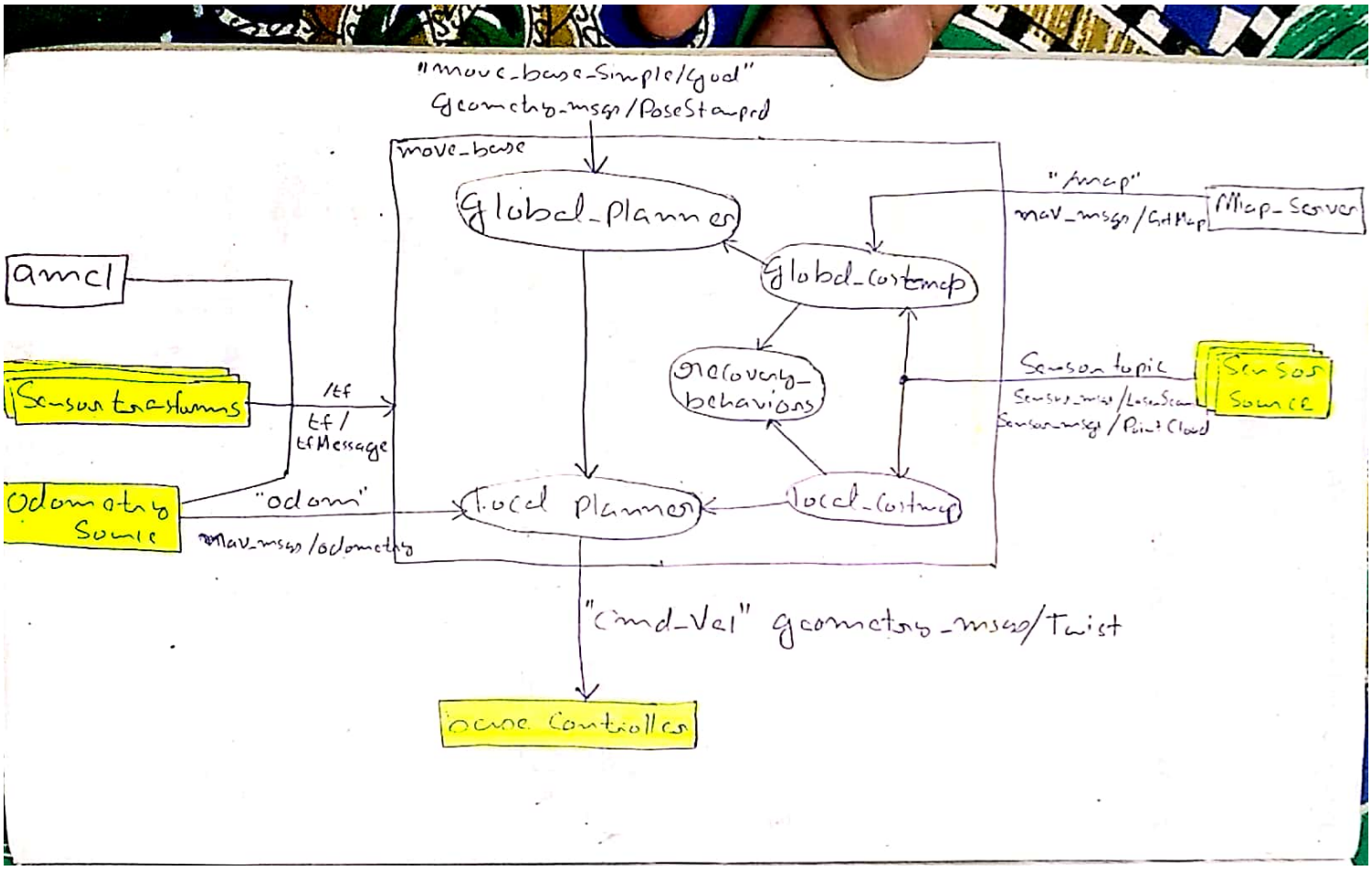
⊙ The optional nodes of the Navigation Stack are amcl and map server, which allow localization of the robot and help to save/load the robot map.

★ Working with Navigation package

★ Understanding the move_base Node

{from package move_base}

⇒ The main function of this package is to move a robot from its current Position to its goal position with the help of other Navigation nodes.



⇒ The move_base node basically is an implementation of SimpleActionServer which takes a goal pose with the message type (geometry_msgs/PoseStamped)

↳ move_base node subscribes the goal from a topic called move_base_simple/goal.

"When move_base node receives a goal pose

↓

It links to Components such as
- global_planner
- local_planner
- recovery_behavior
- global_costmap
- local_costmap

↓

generates the output which is the Command Velocity (geometry_msgs/Twist)

↓

and Send to the base Controller for moving the robot to achive the goal Pose"

⇒ Following is the list of all the packages which are linked by the move-base node:

global-planner

⇒ This package provides libraries and nodes for planning the optimum path from the current position of the robot to the goal position with respect to the robot map.

⇒ This package has implementation of path finding algorithms such as A*, Dijkstra, and so on for finding the shortest path from the current robot position to the goal position.

local-planner

⇒ It takes the odometry and Sensor reading, and send an appropriate velocity command to the robot controller for completing a segment of the global path plan.

⇒ This package is the implementation of the trajectory rollout and dynamic window algorithms.

rotate_recovery

⇒ This package helps the robot to recover from a local obstacle by performing a 360° rotation.

Clear-Costmap-recovery

⇒ Also for recovering from a local obstacle by clearing the costmap by reverting the current costmap used by the Navigation stack to the static map.

Costmap-2D

⇒ The main use of this package is to map the robot environment.

↳ Robot can only plan a path with respect to a map.

⇒ In ROS, we create 2D or 3D occupancy grid maps, which is a representation of the environment in grid of cells.

↓

Each cell has a probability value which indicate whether the cell is occupied or not.

⇒ There are global cost maps for global navigation and local cost map for local navigation.

⇒ Other packages which are interfaced to the move_base node:

⊙ **map_server:**

⇒ Map server package allows us to save and load the map generated by the Cost map 2D package.

⊙ **AMCL**

↳ Method to localize the robot in map

⇒ This approach uses particle filter to track the pose of the robot with respect to the map.

↳ With the help of probability theory.

⇒ On the ROS System, AMCL can only work with maps which were built using laser scans.

⊙ **gmapping**

⇒ Implementation of an algorithm called Fast SLAM which takes the laser scan data and Odometry to build a 2D Occupancy grid map.

★ Over all Working of Navigation Stack

1) Localizing on the map

→ The first step the robot is going to perform is localizing itself on the map.

(AMCL package)

2) Sending a goal and path planning

→ After getting the current position of the robot, we can send a goal position to the move_base node.

→ Move_base node will send this goal position to a global planner

→ Global planner will plan the path from the current robot position to the goal position.

→ global planner sends this path to the local planner, which executes each segment of the global plan.

→ Local planner gets the odometry and the sensor value from the move_base node and find a collision free local plan for the robot.

(local cost map)

3) Collision recovery behavior

→ If the robot is stuck somewhere, the Navigation package will trigger the recovery behaviours nodes.

4) Sending Command Velocity

→ The local planner generates Command Velocity in the form of a twist message, to the robot base Controller.

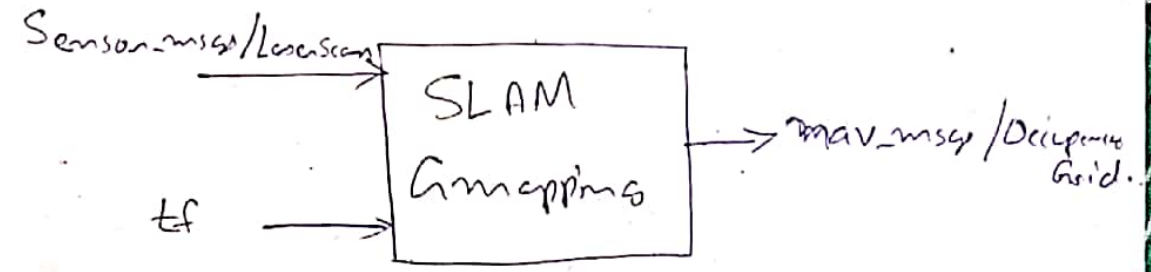
→ The robot base Controller Converts the twist message to the equivalent motor speed.

★ Building a map using SLAM

⇒ slam_gmapping, is the implementation of SLAM which helps to create a 2D occupancy grid map from the laser scan data and the mobile robot pose.

* Creating a launch file for gmapping

⇒ The main task while creating a launch file for the gmapping process is to set the parameters for the slam_gmapping node and the move_base node.



CHAPTER 10

Interfacing I/O Board, Sensor, and Actuators to ROS

7

Interfacing I/O Boards, Sensors, and Actuators to ROS

* Understanding the Arduino-ROS interface

⇒ Arduino boards are powered by Atmel microcontrollers, which are available from 8-bit to 32-bit and clock speed from 8 MHz to 84 MHz.

⇒ Arduino ⇒ Used for quick prototyping of robots.

⇒ Main Application

→ Interfacing sensors and actuators and communicating with PC

- Arduino UNO → Basic robotics & Sensor Interfacing
- Arduino Mega 2560 → Intermediate robotics application level application
- Arduino Due → High end robotics Application

⇒ Most of the communication between PC and I/O board in robots will be through UART protocol.

↳ We can implement our own logic to receive and transmit the data from board to PC and Vice Versa

⇒ The Arduino-ROS interface is a standard way of communication between the Arduino boards and PC.

⇒ We can use the similar C++ APIs of ROS used in PC in Arduino IDE also, for programming the Arduino board.

★ Understanding the rosserial package in ROS

↓

Set of standardized communication protocols implemented for communicating from ROS to character devices such as serial ports and sockets and Vice versa

⇒ The rosserial protocol can convent the Standard ROS messages and service data types to embedded device equivalent data type.

⇒ The Serial data is sent as data Packets by adding header and tail bytes on the packet.

1st Byte	Sync Flag (Value: 0xff)

2nd Byte	Sync Flag / Protocol version

⇒ This byte was 0xff on ROS Groovy and after that it is set to 0xfe

3rd Byte	Message Length (N) – Low Byte

4th Byte	Message Length (N) – High Byte

⇒ This is the length of the packet

5th Byte	Check Sum over message length

⇒ For finding packet Corruption

6th Byte 7th Byte	Topic ID

N Byte	Serialized Message Data

⇒ This is the data associated with each topic.

(N + 8) Bytes	Checksum over Topic ID & Message Data

⇒ Serial data for finding packet Corruption.

⇒ The Checksum of length is computed using the following equation:

$$\text{Checksum} = 255 - \left(\left(\text{Topic ID Low Byte} + \text{Topic ID High Byte} + \ldots \text{data byte value} \right) \% 256 \right)$$

⇒ rosserial_client library helps us to develop ROS nodes from :-

1) Arduino ⇒ rosserial_arduino

2) Embedded Linux Platform ⇒ rosserial_embeddedlinux

3) Windows ⇒ rosserial_windows.

⇒ On the PC side, we need some other Packages to decode the serial messages & Convert to exact topic from the ros_serial_client libraries.

1) rosserial_python
   → The receiving node is completely written in Python

2) rosserial_server
   → Inbuilt functionalities are less Compared to rosserial_python, but it can be used for high Performance application.

★ Installing rosserial packages on Ubuntu

1. Install the rosserial package binaries using apt-get.

```
sudo apt-get install ros-melodic-rosserial
ros-melodic-rosserial arduino
ros-melodic-rosserial-server
```

2. Install latest Arduino IDE

3. Install JAVA runtime support if it is not installed.

```
sudo apt-get install java-common
```

4. Set the location of your sketches in Arduino IDE Preferences.

5. Create a folder name ~~Arduino~~ libraries in the Sketchbook location.

6. Now we can generate ros_lib using a script called make_libraries.py, which is present inside the rosserial_arduino package.

rosrun rosserial_arduino make_libraries.py <Path the library folder>

★ Understanding ROS node APIs in Arduino

⇒ Basic structure of ROS Arduino node.

```
#include <ros.h>
ros::NodeHandle nh;
Void setup()
{
 nh.initNode();
}
Void loop()
{
 nh.spinOnce();
}
```

⇒ We can create the Subscriber and Publisher object in Arduino, similar to the other ROS client libraries.

Defining Subscriber object in Arduino:-

```
ros::Subscriber<std_msgs::String>
    sub("talker", callback);
```

↓

We define a Subscriber which is subscribing a String message, while the callback is the callback function executing when a string message arrives on the talker topic

Defining Publisher object in Arduino

```
ros::Publisher chatter("chatter", &str_msg);
    chatter.publish(&str_msg);
```

⇒ After defining the publisher and the Subscriber, we have to initiate this inside the setup() function using the following lines of code.

```
nh.advertise(chatter);
nh.subscribe(sub);
```

⇒ There are ROS APIs for logging from Arduino.

```
nh.logdebug("Debug Statement")
nh.loginfo("Program info")
nh.logwarn("Warnings")
nh.logerror("Error")
nh.logfatal("Fatalities!")
```

⇒ We can retrieve the current ROS time in Arduino using ROS built in functions such as time and duration.

- Current ROS time

```
ros::Time begin = nh.now();
```

- Convert ROS time in seconds:

```
double secs = nh.now().toSec();
```

- Creating a duration in seconds:

```
ros::Duration ten_seconds(10,0);
```

★ Interfacing Non-Arduino board to ROS

⇒ In such case, we may want to write our own own driver for the board which can convert the serial messages into topics.

CHAPTER 11

ROS Tools

ROS Tools

RViz ⇒ 3D visualization tool

rqt ⇒ Qt based ROS GUI development tool.

rqt_image_view ⇒ Image display tool

rqt_graph ⇒ A tool that visualizes the correlation between nodes and message as a graph.

rqt_plot ⇒ 2D data plot tool

rqt_bag ⇒ GUI based bag data analysis tool.

★ RViz

3D View ⇒ It is the main screen which allows us to see various data in 3D.

Displays ⇒ Form selecting the data that we want to display from various topic.

Menu ⇒ Save or load configuration setting.

Views ⇒ Configures the view point of the 3D view.

★ ROS GUI Development Tool (rqt)

⇒ rqt was developed based on QT, which is a cross-platform framework widely used for GUI Programming, making it very convenient for users to freely develop and add plugins.

CHAPTER 12

Things to know before programming ROS

Things to Know before programming ROS

★ Standard Unit

⇒ The messages used in ROS follows SI Unit.

⇒ It is recommended to use the messages Provided by ROS.

↳ But you can use your own message type if needed.

REP ⇒ ROS Enhancement Proposals.

★ Coordinate representation

⇒ The x, y and z axis in ROS uses right hand rule.

⇒ The front is the positive direction of the x-axis.

↳ (Red)

⇒ The Left side is positive direction of the y-axis.

↳ (Green)

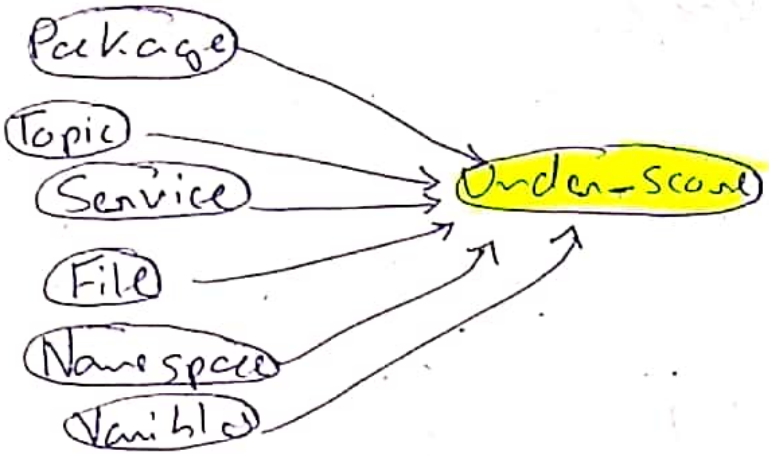
⇒ Upward direction is positive z direction

↓

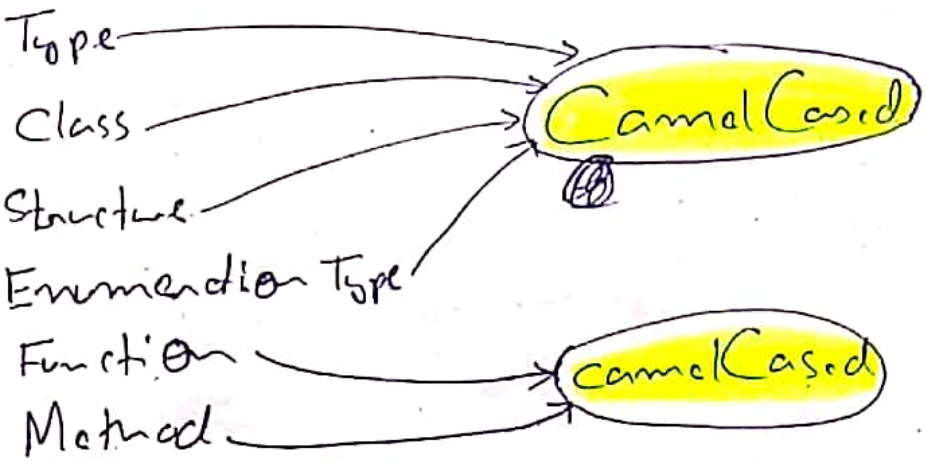
(blue)

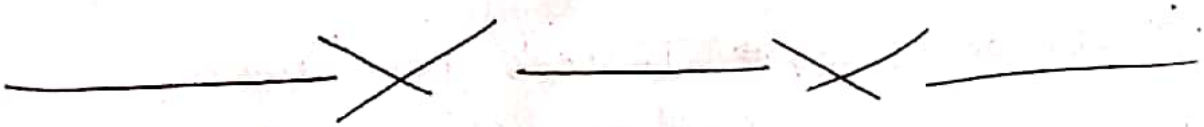
⇒ You can use right hand rule for rotation direction of the robot.

★ Programming Rules



=> However, messages, Service and action file placed in the /msg and /srv folder follows CamelCased style





CHAPTER 13

Robot Sensor Motor

Robot Sensor Motor

★ Robot Package

- Hardware
  - Mechanism, motors, gears, circuit, sensors etc
- Software
  - Firmware
  - Application Software
    - build map, navigate, create motion, perceives environment

⇒ ROS can be classified as application Software and depending on the specialized applications it is classified as
- robot package
- Sensor package
- motor package

★ Sensor package

1D Range Finder ⇒ Infrared distance Sensor.

2D Range Finder ⇒ LiDAR

3D Sensors ⇒ Intel's Real Sense, Microsoft's Kinect, ASUS's Xetion.

Audio/Speech Recognition ⇒ Currently, there are very few areas added to Speech recognition but it seems to be added continuously.

Cameras ⇒ Object recognition, face recognition, Character recognition.

★ Depth Camera

⇒ It can be divided into various types according to the method of acquiring information:

- → ToF (Time Of Flight)
- → Structured Light
- → Stereo Method.

ToF

⇒ An ToF method radiates infrared rays and measure the distance by the time it returns.

⇒ TOF is more expensive than Structured Light.

Structured Light

⇒ Structured Light Projector and camera is used.

⇒ Product now become a History.

Stereo

⇒ It has been researched from much longer time than previous two types.

⇒ Distance is calculated using binocular Parallax like the left and right eyes of a person

* How to use Public Package

↳ ROS Index

↳ Select distribution

↳ Select package

↳ check dependencies

↳ follow the instruction

Scanned by CamScanner

CHAPTER 14

SLAM Theory

SLAM Theory

"SLAM (Simultaneous Navigation and Mapping) means explore and map the unknown environment while estimating the pose of the robot itself by using the mounted sensors on the robot."

⇒ Encoders & IMU are typically used for Pose estimation.

↳ Pose can be estimated without the encoder but only using the inertial sensor.

⇒ The estimated pose can be corrected once again with the surrounding environment information obtained through distance sensor or the camera used when creating the map.

↳ This pose estimation methodology includes Kalman filter, Markov localization, Monte Carlo localization and so on.

⇒ Also, a method of recognizing the environment by attaching markers has been proposed.

* Kalman filter → {Optimal estimation algorithm}

↳ It was used in NASA's Apollo Project and was developed by Dr. Rudolf E. Kalman.

⇒ His filter was a recursive filter that tracks the state of an object in a linear system with noise.

↳ The filter is based on the Bayes Probability which assumes the model and uses this model to predict the current state from the previous state.

↓

Then an error between the predicted value of the previous step and the actual measured present value obtained by measuring instrument is used to perform an update step of estimating more accurate state value.

↓

{The filter repeats above process and increases the accuracy}

⇒ However, the Kalman filter only applies to linear systems.

⇒ EKF (Extended Kalman Filter) modified from Kalman filter are widely used.

⇒ There are also many Variants such as UKF (Unscented Kalman Filter) which improved the accuracy of EKF.

⇒ Fast Kalman filter improved speed and these are still being researched today.

★ Particle Filter

⇒ Particle filter is most popular algorithm in object tracking.

→ Example Monte Carlo localization using particle filter.

⇒ Because robots and sensors are also nonlinear, particle filters are often used for pose estimation.

⇒ Partical filter is a technique to predict through simulation based on trial and error method.

⇒ When using SLAM, the robot's odometry value and the measurement values using the distance sensor are used to estimate the robot's current pose.

CHAPTER 15

Navigation Theory

Navigation Theory

★ Costmap

⇒ Cost map can be divided into two:

① Global Costmap

⇒ Sets up path plan for navigating in the global area of the fixed map.

② Local Costmap

⇒ Used for path planning and obstacle avoidance in the limited area around the robot.

⇒ Although purposes are different, both Costmaps are represented in the same way.

⇒ The Cost map is expressed as a value between 0 and 255

000 ⇒ Free area where robot can move freely.

001 – 127 ⇒ Area of low Collision Probability.

128 – 252 ⇒ Area of high Collision Probability

253 – 254 ⇒ Collision area

255 ⇒ Occupied area where robot cannot move.

★ AMCL

→ (Adaptive Monte Carlo Localization)

⇒ The ultimate goal of Monte Carlo Pose estimation (MCL) is to determine where the robot is located in a given environment.

→ (i.e. we must get x, y and θ of the robot on the map)

★ Dynamic Window Approach (DWA)

⇒ DWA is a popular method for obstacle avoidance planning and avoiding obstacles.

⇒ This is a method of selecting a speed that can quickly reach a target point while avoiding the obstacles.

⇒ In DWA the objective function G is used to calculate the translational velocity v and the rotational velocity ω and maximizes the objective function which considers the direction, velocity and collision of the robot.

CHAPTER 16

Ros Developers Guide

ROS developer's guide

1. Source control
   - Git

⇒ Add to source control only the minimal set of manually written source & build files that are necessary to build your package.

- Dont add machine generated files such as:
  - Objects (.o)
  - libraries (.so)
  - auto-generated Configure script.
- Dont add large binary files:
  - Upload them to a web server and download them in your build file.

⇒ Commit your code early & often.

⇒ Try to keep commits focused on a particular change instead of lumping multiple changes together.

⇒ Give an informative message with each commit.

⇒ Dont break the build.

2. Bug tracking

⇒ For package hosted on GitHub this is commonly the issue tracker of the repository.

⇒ The maintainer will assign milestone to each reported issue.

⇒ To raise an issue:

(1) Be as descriptive as possible.

→ Include instructions for reproducing the bug.

3. Code layout

⇒ Packages can be collected in a single repository.

⇒ For GitHub it is recommended to Create a README.md at the root of your repository to explain to users what to find in the repository.

⇒ It is recommended to link to the Package documentation on the ROS wiki for the contained package.

4.

⇒

5

6

4. Packaging

⇒ The ROS package and build system relies on package.xml files.

- Every package must have a package.xml file, located in the package's top directory.
- At a minimum, the package.xml must contain:
  - description
  - author
  - license

5. GUI toolkits

- Use rqt for any new GUI development.

{Qt-based GUI framework for ROS}

6. Building

⇒ The basic build tool is CMake.

- Every package that has a build step must have a CMakeList.txt file in the package's top directory.
- Packages that don't have build steps don't need any build files.

7. Licensing

~~⇒ We prefer permissive~~

⇒ The preferred license for the Project is the BSD license.

↳ Whenever possible, new code should be licensed under BSD.

⇒ The full text statements of all licenses used in the project should be placed in the LICENSES directory at the top of the repository.

⇒ Every source file should contain a commented License Summary at the top.

(8) Copyright

⇒ Under the Berne Convention, the author of a work automatically holds copyright, with or without a formal statement to that effort.

↳ However, making copyright explicit is helpful in long-term project management

⇒ Each source file should contain a commented copyright line at the top

eg: Copyright 2008 Jim Bob

↓ statement

{This is usually directly above the license}

⇒ If you work for yourself, then you own the copyright.

⇒ If you work for someone else, then your employer owns the copyright.

⇒ The most popular open source license has an important aspect in common.

↳ The Open Source Initiative (OSI) has approved them.

formed in 1998

Applying a License to your open source project

↳ You need to add a LICENSE, LICENSE.txt or LICENSE.md in the root directory of your repository.

---

⑨ Debugging

⇒ Generally used debugging tools used in ROS:

- GDB
- Oprofile
- Valgrind

10) Testing

⇒ We use two level of testing

- Library ⇒ At the library level, we use standard unit-test frameworks.
  - → In C++, we use gtest
  - → In Python, we use unittest.
- Message ⇒ At the message level, we use rostest to set up a system of ROS nodes, run a test node, then tear down the system.

11) Documentation

⇒ All code should be documented according to QA Process. Including
- → All externally visible code-level API must be documented.
- → All externally visible ROS-level APIs (topic, services, Parameters) must be documented.

12) Releasing

⇒ The standard process for releasing code to the ROS community is described on the bloom documentation.

13)

14)

13) Standardization

⇒ Code should use ROS services, follow guidelines for their use:-

- Use rosout for printing message
- Use the ROS clock for time based routines.

14) Deprecation

⇒ As soon as there are users of your code, you have a responsibility not to pull the rug out from under them with sudden breaking changes.

- Instead use the process of deprecation, which means making a feature or component as being no longer supported, with a schedule for its removal.

⇒ Deprecation can happen at multiple levels:

① API feature
- Mark it as deprecated in the API documentation; with Doxygen.

② Package
- Mark it deprecated in Wiki documentation.

15) Large data files & Large test files

⇒ Large data files should be hosted in a public web hosting site.

↳ {anything over 1MB}

⇒ To download this file for building use the catkin_download_test_data.

CHAPTER 17

package.xml

Package.xml

⇒ Must be included with any catkin - compliant package's root folder.

↳ This file defines properties about the package and dependencies on the other catkin packages.

⇒ Others depend on this information to install the software they need for using your package.

↳ You may be able to run test without this file.

★ Format 2 (Recommended)

Each package.xml file has the <Package> tag as the root tag in the document.

* <package format="2">

Required Tags

There are minimal set of tags that need to be nested within the <package> tag to make the package manifest complete.

<name>
<version>
<description>
<maintainer>
<license>

Dependencies

⇒ Package can have six types of dependencies.

① Build Dependencies

⇒ Specify which package are needed to build this package.

② Build Expart Dependencies

⇒ which packages are needed to build libraries against this Package.

③ Execution Dependencies

⇒ Specify which package are needed to run code in this package

④ Test Dependencies

⇒ Specifies only additional dependencies for unit test.

⇒ They should never duplicate any dependencies already mentioned in as build or run dependencies.

⑤ Build Tool Dependencies

⇒ Specifies build tools which this Package to build itself.

⑥ Documentation tool Dependencies

⇒ Specifies tools that this package need to generate documentation.

⇒ The 6 type of dependencies are specified using the following respective tags.

<depend>
↳ specifies that a dependency is a build, export and execution dependencies.

⑤ <buildtool_depend>
① <build_depend>
② <build_export_depend>
③ <exec_depend>
④ <test_depend>
⑥ <doc_depend>

⇒ All packages must have at least one dependency, a build tool dependency on catkin.

Metapackage

⇒ It is always convenient to group multiple packages as a single logical package.

⇒ A metapackage is a named package with the following export tag in the package.xml.

<export>
<metapackage/>
</export>

⇒ meta package requires <build tool-depend> on catkin.

⇒ Metapackage has a required, boilerplate CMakeLists.txt file:

```
cmake_minimum_required (VERSION 2.8.3)
Project (<Package-Name>)
find_package (catkin REQUIRED)
catkin_metapackage()
```

Additional tags

<URL> → URL for information on the package, typically on wiki page on ros.org

<author>

CHAPTER 18

CMakeList.txt

CMakeList.txt

⇒ The file CMakeLists.txt is the input to the CMake build system for building software packages.

↳ It describe how to build the code and where to install it.

* Overall Structure and Ordering

⇒ The order in the Configuration DOES Count.

1. Required CMake Version (cmake_minimum_required)
2. Package Name (project())
3. Find Other CMake/Catkin packages needed for build (find_package())
4. Enable Python module support (catkin_python_setup())
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())

* CMake Version

cmake_minimum_required
(VERSION 2.8.3)

* Package name

Let say name of project is, orobot_brain

⇒ Project (orobot_brain)

⇒ You can reference the project name anywhere later in the CMake script by using the Variable ${PROJECT_NAME}

* Finding Dependent CMake Packages

⇒ There is always at least one dependency on catkin:

find_package (catkin REQUIRED)

⇒ You should ~~always~~ only find_package components for which you want build flags. You should not add runtime dependencies

What does find_package() Do?

⇒ If a package is found by CMake through find_package, it results in the creation of several CMake environment variables that give information about the found package.

⇒ The names always follows the convention of

<PackageName>_<Property>

⇒ <NAME>_FOUND

↳ Set to true if library is found otherwise false.

⇒ <NAME>_INCLUDE_DIRS or <NAME>_INCLUDES

↳ This includes path exported by the package.

⇒ <NAME>_LIBRARIES or <NAME>_LIBS

↳ The libraries exported by the package.

Boost

⇒ If you are using C++ and Boost, you need to invoke find_package() on Boost and specify which aspect of Boost you are using as Components.

* catkin_package()

↳ Catkin-provided CMake macro.

⇒ This function must be called before declaring any targets.

⇒ The function has 5 optional argument:

INCLUDE_DIRS
LIBRARIES
CATKIN_DEPENDS
DEPENDS
CFG_EXTRAS

* Specifying Build Targets

⇒ Build targets can take many forms, but usually they represent one of two possibilities:

- Executable Target
  ↳ Programs we can run
- Library Target
  ↳ Libraries that can be used by executable targets at build and/or runtime.

Target Naming

⇒ One can have a target renamed to something else using the set_target_properties() function:

Example

```
set_target_properties (rviz_image_view
                       PROPERTIES OUTPUT_NAME
                       image_view
                       PREFIX "")
```

This will change the ~~name~~ of the target rviz_image_view to image_view in the build and install output

Custom output directory

⇒ While the default output directory for executables and libraries is usually set to a reasonable value it must be customized in certain case.

Example: Library containing python bindings must be placed in a different folder to be importable in Python.

set_target_properties (Python_module_library
PROPERTIES LIBRARY_OUTPUT_DIRECTORY
${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE
_PYTHON_DESTINATION})

Include Path and Library Path

⇒ Prior to specifying targets, you need to specify where resources can be found for said target.

{header files & libraries}

@ include_directories()

include_directories(include ${catkin_INCLUDE
_DIRS})

@ link_directories()

↳ This is not recommended.

link_directories(~/my_libs)

Executable Targets

⇒ To specify an executable target that must be built, we must use the add_executable() CMake function.

Example:

add_executable (my_Program src/main.cpp src/sum_file.cpp)

Library Target

⇒ The add_library( ) CMake function is used to specify libraries to build.

⇒ By default catkin builds shared libraries.

Example

add_library(${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})

target_link_libraries

- Use the target_link_libraries() function to specify which libraries an executable target link against.

target_link_libraries (<executable Target Name> <lib1> <lib2> ... <libN>)

★ Messages, Services and Action Targets

⇒ Message (.msg), Services (.srv) and action (.action) files in ROS require a special Preprocessor build step before being built and used by ROS package.

⇒ The point of these macros is to generate programming language-specific files so that one can utilize it in their programming language of choice.

⇒ There are three macros provided to handle messages, services, and actions respectively:

```
add_message_files
add_service_files
add_action_files
```

⇒ These macros must be followed by a call to the macro that invokes generation:

```
generate_message()
```

CHAPTER 19

Doxygen

Doxygen

⇒ It is a tool for automatic documentation generation based on commented code.

- It is a program that can look into your source files, extract comments and make useful documentation.
- Works with both C++ & Python.
- Generate html code (several web pages)
  - Can also generate latex or RTF.

★ How to use

1. Setup a configuration file
   - Make a doc folder in the Project root and then enter: doxygen -g
2. Edit the Configuration file
3. When comments in the code change, run doxygen in the folder with Doxyfile.

★ Commenting your code

⇒ If you place comments before the functions and variables use:

- /** */ OR /*! */ for multiple comments OR
- //! OR /// for single line comments.

⇒ If you place Comments after functions or variables use:

- /**< */ OR /*! < */ for multiline comments OR
- //! < OR /// < for single line comments.

⇒ @ or \ preceded text are seen as commands by doxygen

★ Working with Doxygen

sudo apt-get install doxygen

\author
\Version
\date
\bug
\warning
\copyright
\mainpage
\section
\subsection

⇒ Empty line to make seperation between Contents.

\brief
\Param
\details

INPUT = ../Src ../include → {directory for all the source file}

FILE_PATTERNS = *.cpp *.h → {All the files it should look for}

EXTRACT_ALL = YES

GENERATE_HTML = YES

GENERATE_LATEX = NO

PROJECT_NAME = "My-Project"

CHAPTER 20

Working with Pluginlib, Nodelets and Gazebo Plugins

Working with Pluginlib, Nodelets and Gazebo Plugins

* Understanding pluginlib

⇒ Plugins are modular piece of software which can add a new feature to the existing ~~software~~ software application.

↳ Using this method, we can extend the capabilities of software to any level.

⇒ When we are going to build a complex ROS based application for a robot, plugins will be a good choice to extend the capabilities of the application.

⇒ The ROS system provides a plugin framework called pluginlib to dynamically load / unload plugins, which can be a library or class.

* Creating plugins for the calculator application using pluginlib

⇒ The aim of this example is to show how to add new feature to calculator without modifying main application code.

{ catkin_create_pkg pluginlib_calculator
pluginlib roscpp std_msgs }

Step 1: Creating calculator_base header file

⇒ main purpose of this file is to declare functions / methods that are commonly used by the plugins.

namespace calculator_base

↳ class ⇒ calc_functions

↳ Member fuctions

↳ get_numbers

↳ operation

Step 2: Creating calculator_plugins header file

⇒ Main functions of this * calculator Plugins, which are named as Add, Sub, Mul, and Div.

{ file is to define complete function of the }

Step3 - Exporting plugins using calculator_plugins.cpp

⇒ In order to load the class of plugins dynamically, we have to export each class using a special macro called PLUGINLIB_EXPORT_CLASS.

```
#include <pluginlib/class_list_macros.h>

PLUGINLIB_EXPORT_CLASS(calculator_plugins::Add, calculator_base::calc_function);
```

Step4 - Implementing plugin loader using calculator_loader.cpp

⇒ The plugin loader node loads each plugin and inputs the number to each plugin and fetch's the result from the plugin.

```
#include <boost/shared_ptr.hpp>
#include <pluginlib/class_loader.h>

pluginlib::ClassLoader<calculator_base::calc_functions>
calc_loader("pluginlib_calculator", "calculator_base::calc_functions");
```

boost:: shared_ptr <Calculator_base ::
calc_functions> add = Calc_loader.
CreateInstance ("pluginlib_Calculator/Add");

Step 5: Creating plugin description file : Calculator_plugin.xml

<library path="lib/libpluginlib_Calculator">

<class name="pluginlib_Calculator/Add"
type="Calculator_plugins :: Add"
base_class_type="Calculator_base:: calc_function">

<description> This is add plugin. </description>

</class>

.
.
.

</library>

Step 6: Registering plugin with the ROS Package system

⇒ For pluginlib to find all plugins based Packages in the ROS System, we should export this plugin description file inside Package.xml.

⇒ If we do not include this plugin, the ROS system won't find the plugins inside the package.

```
<export>
<plugimlib_Calculator plugin="${Prefix}
/calculator_plugins.xml" />
</export>

<build_depend> plugimlib_calculator
</build_depend>
<exec_depend> plugimlib_calculator
</exec_depend>
```

Step 7 - Editing the CMakeList.txt file

```
add_library (plugimlib_calculator
             src/calculator_plugins.cpp)
target_link_library (plugimlib_calculator
                     ${catkin_LIBRARIES})

add_executable (Calculator_loader
                src/calculator_loader
                .cpp)
target_link_libraries (Calculator_loader ${catkin
                                          _LIBRARIES})
```

Step 8: Querying the list of plugins in a Package

rospack plugins --attrib=plugin pluginlib_calculation

Step 9: Running the plugin loader

rosrun pluginlib_calculation calculation_loader

★ Understanding ROS nodelets

⇒ Nodelets are a type of ROS node that are designed to run multiple nodes in a single process, with each node running as a thread.

↳ These threaded nodes can communicate with external nodes too.

⇒ In nodelets, we dynamically load each class as a plugin, which has a separate namespace.

⇒ Nodelets are used when the volume of data transferred between nodes are very high.

example ⇒ 3D sensor

* Creating a Nodelet

"We are going to Create a basic nodelet that Can Subscribe a String topic called /msg_in and Published the Same String on the topic /msg_out"

Step1- Creating a package for nodelet

Step2- Creating hell_world.cpp nodelet

⇒ We should include class_list_macro.h and nodelet.h to access pluginlib API's and Nodelets APIs.

```
namespace nodelet_hello_world
{
Class Hello : public nodelet::Nodelet
{
```

⇒ All nodelet Class should inherit from the nodelet base class and be dynamically loadable using pluginlib.

```
Virtual void onInit()
{
ros::NodeHandle privateNh = getPrivateN
                            odeHandle();
```

```
NODELET_DEBUG ("Initialized the Nodelet")
pub = private_nh.advertise<std_msgs::String>
                          ("msg_out", 5);
sub= private_nh.subscribe("msg_in", 5, &Hello::
                          callback, this);
}
```

⇒ This is the initilization function of a nodelet.

↳ Inside this function we are creating a node handle object, topic publisher and Subscriber on the topic msg_out and msg_in respectively.

```
PLUGINLIB_EXPORT_CLASS
(nodelet_hello_world::Hello, nodelet::Nodelet);
```

⇒ Here we are exporting the Hello as a plugin for the dynamic loading.

Step 4 – Creating plugin description file

hello_world.xml

```
<library path="lib/nodelet_hello_world">
  <class name="nodelet_hello_world/
    Hello" type="nodelet_hello_world::HELLO"
    base_class_type="nodelet::Nodelet">
  <description>
  A node to publish a message
  </description>
  </class>
</library>
```

Step 5 – Adding the export tag in package.xml

We need to add the export tag in package.xml and also add build and run dependencies.

Step 6 – Editing CMakeList.txt

```
add_library (nodelet_hello_world
             src/hello_world.cpp)
target_link_libraries (nodelet_hello_world
                       ${catkin_LIBRARIES}
                       )
```

Step 7 - Building and running nodelets

⇒ The first step in running nodelets is to start the **nodelet manager**.

↓

It is a C++ executable program, which will listen to the ROS service and dynamically load nodelets

⇒ To start nodelet manager:

```
rosrun nodelet nodelet manager __name:=
                                nodelet_Manager
```

⇒ After launching the nodelet manager, we can start the nodelet by using the following Commands:

```
rosrun nodelet nodelet load nodelet_hello
    _world/Hello node_manager
    __name:=nodelet.1
```

↓

⇒ Creates instance of nodelet/hello_world /Hello nodelet with a name nodelet1.

Step 8 – Creating launch files for nodelets

```
<launch>

<node pkg="nodelet" type="nodelet"
 name="standalone_nodelet"
  args="manager" output="screen"/>


<node pkg="nodelet" type="nodelet"
 name="test" args="load nodelet_hello
 _world/Hello standalone_nodelet"
 output="screen"/>

</launch>
```

★ Gazebo Plugins

⇒ Gazebo plugins help us to control the robot models, sensors, world properties, and even the way Gazebo runs.

⇒ We can Mainly classify the Plugins as follows:

- **World plugin**
  - ↳ using this we can control the properties of a specific world in Gazebo.

- **Model plugin**
  - Parameters such as joint state of the model, control of the joints and so, on can be controlled using this plugin.
- **Sensor plugin**
  - These are for modelling sensors such as Camera, IMU, and so on in Gazebo.
- **System plugin**
  - Control system related function in Gazebo.
- **Visual plugin**
  - Visual properties of any Gazebo component can be accessed and controlled using the visual plugin.

⇒ Gazebo plugins are independent of ROS and we don't need ROS libraries to build the plugin.

CHAPTER 21

Writing ROS Controllers and Visulization Plugins

6

Writing ROS Controllers and Visulization Plugins

⇒ We will continue with pluginlib based concepts such as ROS controllers and RViz plugins.

⇒ The main set of packages used to develop a controller generic to all robot is contained in the ros_control stack.

→ Rewritten version of pr2_mechanism.

⇒ Useful packages that help us to write robot controllers:

- **ros_control**

  → This package takes as input the joint state data directly from the robot's actuators and the desired set point, generating the output to send to send to its motors.

  → Output is usually represented by the joint position, velocity or effort.

- Controller_manager
  - Control manager can load and manage multiple controllers and can work them in a real-time compatible loop.
- Controller_interface
  - This is the controller base class package from which all custom controllers should inherit the controller base class.
  - The controller manager will only load the controller if it inherits from this package.
- hardware_interface
  - This package represents the interface between the implemented controller and hardware of the robot.
- joint_limits_interface
  - This package allows us to set joint limits to safely work with our robot.

- realtime_tools
  ⇒ Contains set of tools that can be used from a hard real-time thread.

★ Understanding ros_control packages

@ The controller_interface package

⇒ The basic ROS low level controller that we want to implement must inherit a base class called

controller_interface :: Controller

↓

This is a base class containing four fundamental functions

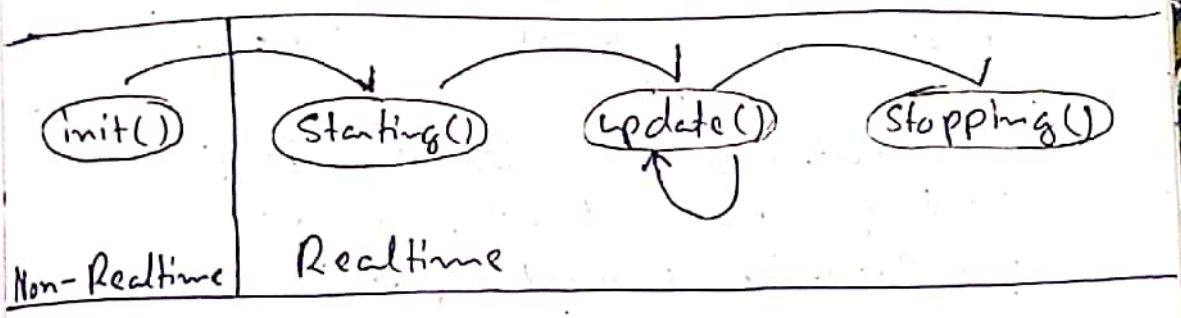
{ init(), start(), update() and stop() }

⇒ The basic structure of the controller class is given as follows:

```
namespace controller_interface
{
  class_Controller
    {
    Public:
      virtual bool init (hardware_Interface
                         *robotHW, ros:: NodeHandle &nh);
```

```
    virtual void starting();
    virtual void update();
    virtual void stopping();
};
}
```

⇒ The workflow of the controller class is shown as follows:



Initializing the controller

⇒ The first function executing when a controller is loaded is init().

↓

{It just initializes the controller}

hardware_interface ⇒ This variable represents the specific hardware interface used by the controller to develop.

↳ ROS contains a list of already-implemented hardware interfaces, such as:

- → Joint Command Interface (effort, velocity and position)
- → Joint State Interface
- → Actuator State Interface

↳ We can even create our own hardware interface.

ros::NodeHandle ⇒ The controller can read the robot configuration and even advertise topics using this NodeHandle.

Starting the ROS Controller

⇒ starting() method executes once just before running the controller.

⇒ The controller can also call the starting() method when it restarts the controller without uploading it.

Updating the ROS controller

update() method, by default, executes the code inside it at a rate of 1000 Hz.

Stopping the Controller

stopping() method will be called when a controller is stopped.

@ The controller_manager

⇒ The controller_manager package can load and unload the desired controller.

⇒ The controller_manager also ensures that the controller will not set a goal value that is less than or greater than the safety limits of the joints.

⇒ The controller_manager also publishes the states of the joint in the /joint_state topic at a default rate of 100 Hz.

★ Writing a basic joint controller in ROS

Step 1: Creating the controller package with necessary dependencies

```
catkin_create_pkg my_controller roscpp
                  pluginlib controller_interface
```

(2) Write Controller code in C++

(3) Register or export the C++ class as plugin.

(4) Define plugin definition in a XML file

(5) Edit the CMakeList.txt and package.xml files for exporting the plugin

(6) Write the configuration for our controller

(7) Load the controller using the controller manager.

* Understanding the ROS visualization tool (RViz) and its plugins

(The standard method to build an ROS plugin is applicable for this plugin too)

⇒ RViz is written using a GUI framework called Qt.

CHAPTER 22

Working with ROS actionlib

Working with ROS actionlib

⇒ In ROS services, the user implements a request/reply interaction between two nodes, but if the reply takes too much time on

↓

the service is not finished with the work

↓

We have to ~~wait~~ wait until it completes, blocking the main application.

⇒ Actionlib package provides a standard way to implement these kind of Preemptive tasks.

⇒ Actionlib is highly used in robot arm navigation and mobile robot navigation.

⇒ The action specification is stored inside the action file having an extension of .action.

⇒ The action file has the following parts:

1) Goal ⇒ The action client can send a goal that has to be executed by the action server.

2) Feedback ⇒ Feedback is simply giving the progress of the current operation inside the callback function.

3) Result ⇒ After completing the goal, the action server will send a final result of completion

↓

{ It can be the computational result or an acknowledgment. }

CHAPTER 23

Applications of topic, service and actionlib

Applications of topics, services and actionlib

* Topics ⇒ Streaming continuous data flow, such as sensor data.

* Services ⇒ Execute procedures and terminate quickly.

* Actionlib ⇒ Execute long and complex action managing their feedback.

CHAPTER 24

Moving Robot Joints Using ROS Controllers in Gazebo

Moving robot joints using ROS Controllers in Gazebo

⇒ For each joint we need to attach a controller that is compatible with the hardware interface mentioned inside the transmission tag.

⇒ The ROS controller mainly consists of a feedback mechanism that can receive a set point and control the output using the feedback from the actuator.

⇒ ROS Controller interacts with the hardware using hardware interface.

⇒ The hardware interface is a Software representation of the robot and its abstract hardware.

⇒ The resources of the hardware interface are actuators, joints and sensors.

↳ Some resources are read only and some write compatible.

ros_control

"A set of packages that includes controller interfaces, controller managers, transmissions and hardware_interface."

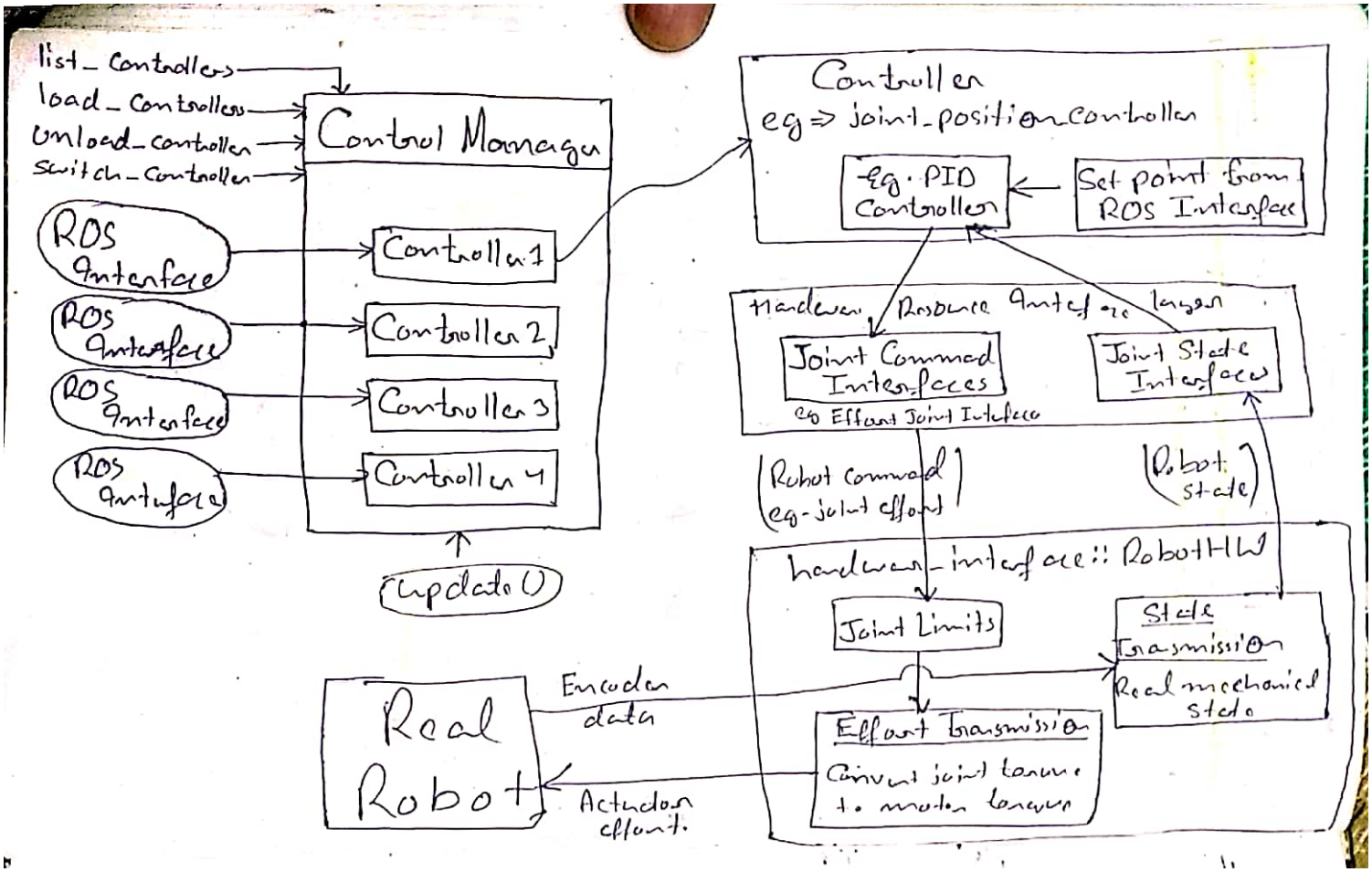
⇒ The ros_control packages takes as input the joint state data from your robot's actuators encoders and an input set point.

↳ It uses a generic control loop feedback mechanism, to control the output.

↳ {typically Position Velocity or effort}

★ Data flow of ros_control and Gazebo

⇒ "In addition to the transmission tags, a Gazebo plugin needs to be added to your URDF that actually parses the transmission tags and loads the appropriate hardware interfaces and control managers"



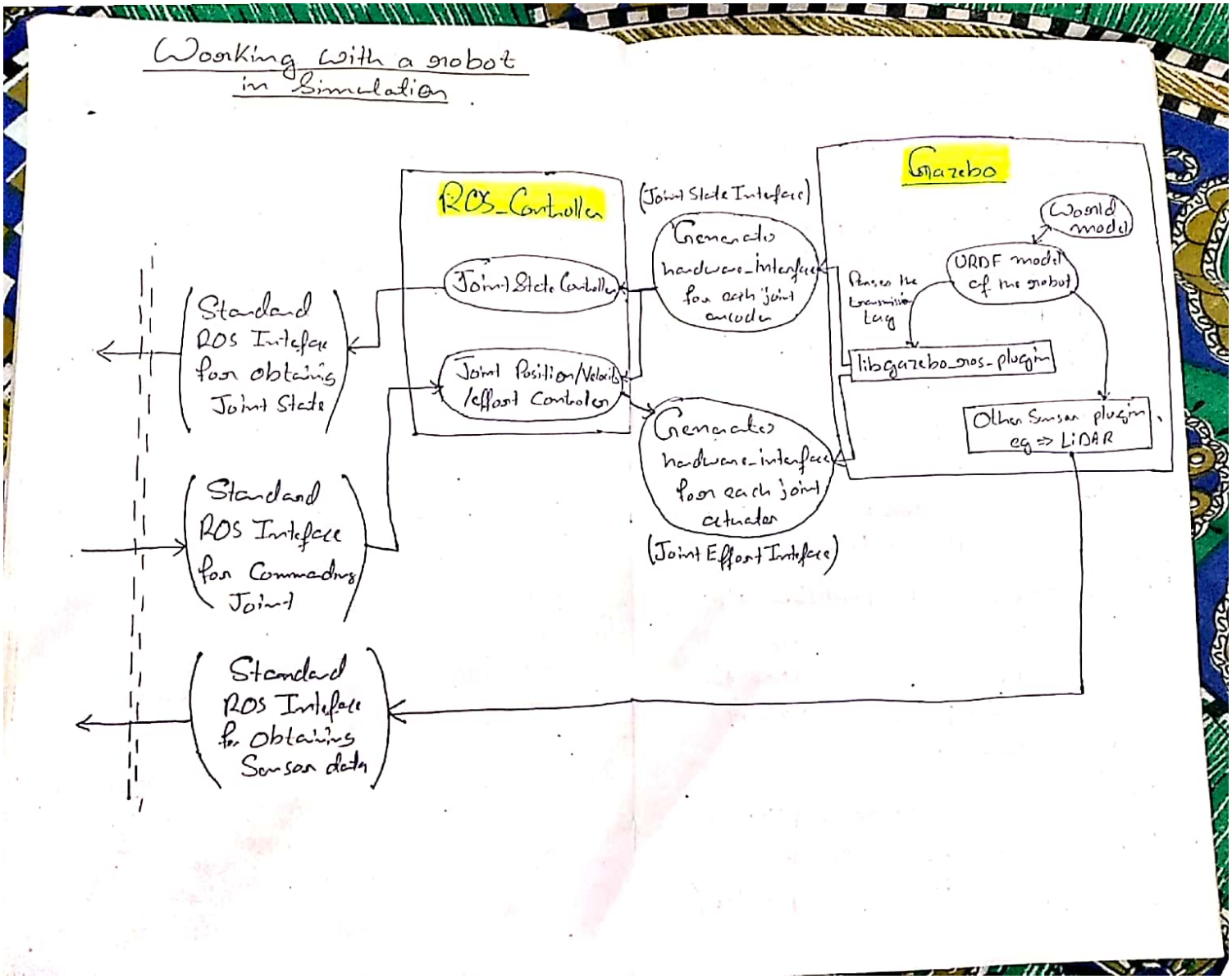
hardware_interface

⇒ When you make your robot support one or more of the standard interfaces, you will be able to take advantage of a large library of Controllers that work on the standard interface.

CHAPTER 25

Working with a robot in Simulation

Working with a robot in Simulation



★ Work flow for working with Robot in Simulation

① Build a detailed URDF model of the robot.

(With all the inertial detail, material property (Specially for parts which will be in contact), Joint properties for non-rigid joint and necessary plugin for ros_control and Sensors attached)

② Write ROS controller for controlling each joint.

(Try to use standard ROS controller whenever possible to save time.)

③ Use robot state publisher to publish tf data.

④ Write Application Software for Commanding your robot as a whole.

{Try to use standard package such as Navigation stack and Moveit etc. whenever possible}

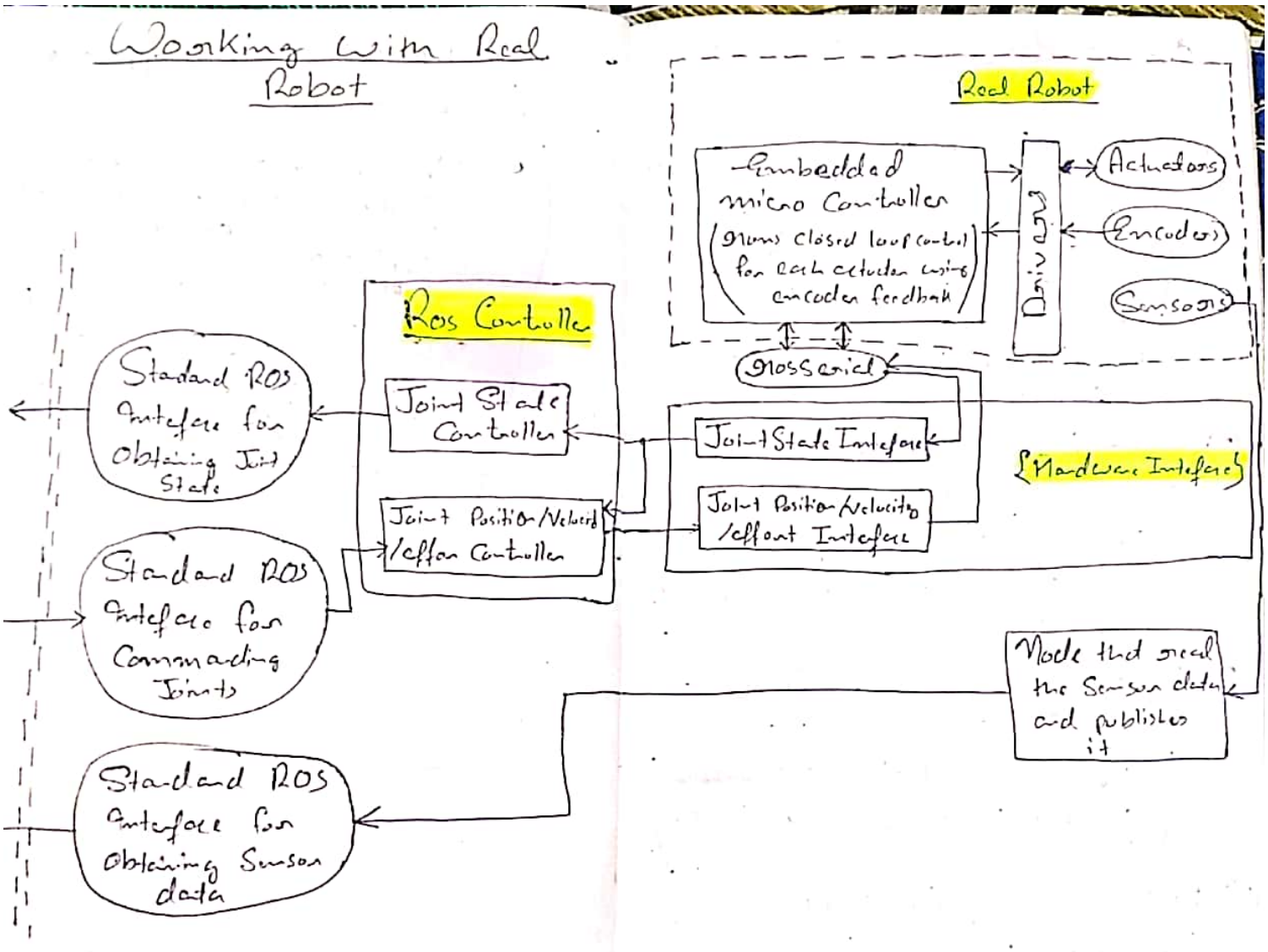
(5) Build UI (User Interface) application for your robot.

eg ⇒ Voice based communication, so the robot understand what user is saying and infer which application software to call and with what goal.

⇒ Or Simple teleoperation Control of the robot using Key board & display based interface.

CHAPTER 26

Working with real Robot



★ Workflow for Working with Real Robot

(5)

(6)

① Write the firmware for embedded micro controller to run close loop control of all the actuator, interface all the basic sensors (i.e. IMU) and perform Serial Communication ~~with~~ with the microprocessor using standard protocol established by rosserial.

② Write the hardware interface for the robot.

(Try to select standard hardware interface available whenever possible to save time.)

③ Write ROS Controllers for controlling each joints.

(Try to use standard ros controller available whenever possible to save time.)

④ Build a Simplified URDF model of your robot which will be then used by robot State publisher to publish tf data.

(5) Write Application Software for Controlling your robot as a whole

(6) Build UI application for your Robot.

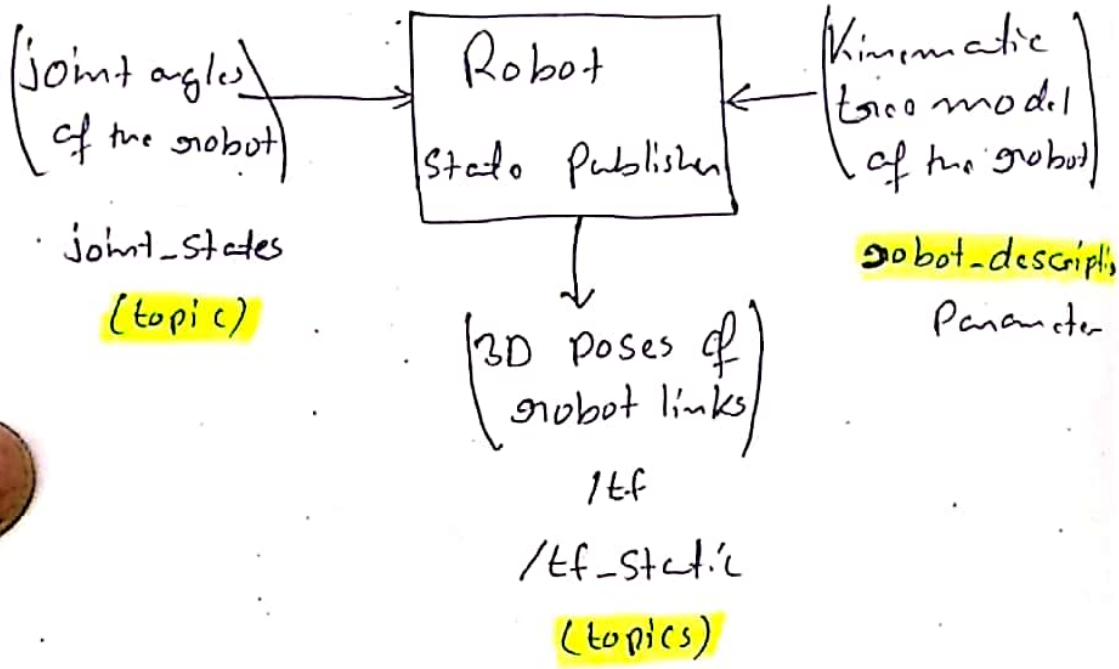
CHAPTER 27

Robot State Publisher

robot_state_publisher

⇒ This package allows you to publish the state of a robot to tf.

⇒ It takes joint angles of the robot as input and publish the 3D poses of the robot links using Kinematic tree model of the robot.



* API

Subscribed topic

→ joint_states (sensor_msgs/JointState)

Parameters

→ robot_description

(The Urdf xml robot description)

*

*

→ tf_prefix (string)

(for namespace-aware publishing of transform.)

→ publish_frequency (double)

(Publish frequency of state publisher, default: 50Hz)

→ ignore_timestamp (bool).

(If true, ignore the publish_frequency and timestamp of joint_states and publish a tf for each of the recived joint state.
Default is false)

→ use_tf_static (bool)

(Set whether to use the /tf_static latched static transform broadcaster
Default: true)

CHAPTER 28

Navigation Tutorial

Odometry

⇒ It is use of data from motion sensors to estimate change in position over time.

{ Position relative to a starting location }

Navigation

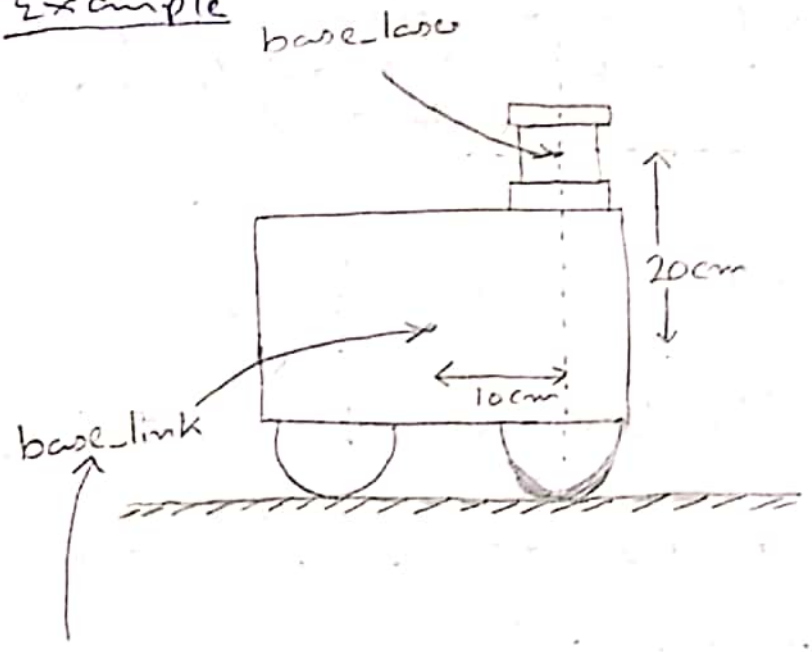
1. Setting up your robot using tf

★ Transform Configuration

⇒ Many ROS packages require the transform tree of a robot to be published using the tf software library.

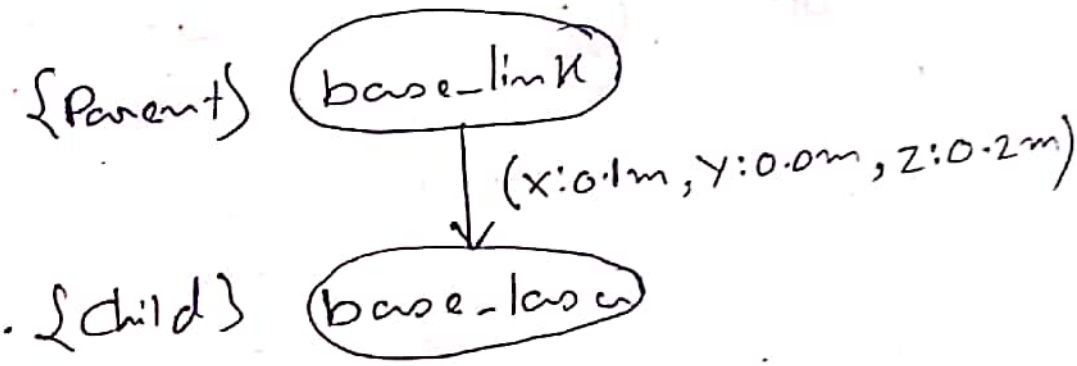
→ { defines offsets in terms of both translation and rotation between different coordinate frames }

Example



For navigation its important that this be placed at the rotational center of the robot

⇒ Each node in the transform tree corresponds to a coordinate frame and each edge corresponds to the transform that needs to be applied to move from the current node to its child.



⇒ Parent and child distinction is important because tf assumes that all transform moves from parent to child.

* Writing Code

* Broadcasting a Transform

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

tf::TransformBroadcaster broadcaster;

while (n.ok()) {
  broadcaster.sendTransform(
    tf::StampedTransform(
      tf::Transform(tf::Quaternion
      (0,0,0,1), tf::Vector3(0.1,0.0,0.2)),
      ros::Time::now(), "base_link",
      "base_laser"));

  r.sleep();

}
```

{Child node}

{Parent node}

* Using a Transform

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

⇒ It includes files that we need to create a tf::TransformListener.

↳ TransformListener object automatically subscribe to the transform message topic over ROS and manages all transform data comming in over the wire.

```
void transformPoint (const tf::Transform
                     Listner & listner)
```

⇒ We'll create a function that takes a point in the "base_laser" frame and transforms it to the "base_link" frame.

```
listner.transformPoint ("base_link"
                        , laser_point, base_point);
```

⇒ Give the data in laser_point to base_point with respect to "base_link" frame.

★ Building the Code

CMakeLists.txt

add_executable (tf_broadcaster
src/tf_broadcaster.cpp)

add_executable (tf_listener src/tf_listener.cpp) ★

target_link_libraries (tf_broadcaster ★
${catkin_LIBRARIES})

target_link_libraries (tf_listener
${catkin_LIBRARIES})

2.

Basic Navigation Tuning Guide

* Range Sensors
  - Ensure Laser scan data is correct and is comming at expected rate.

* Odometry
  {AMCL}

* Localization

⇒ First run either gmapping or Karto and joystick the robot around to generate a map.

⇒ Then use this map with AMCL and make sure that the robot stays localized.

* The Costmap
* Local & Global Planner

3. Setup and Configuration of the Navigation stack on a robot

⇒ The navigation stack requires that the robot be publishing information about the relationships between Coordinate frames using tf.

⇒ The navigation stack assumes that these sensors are publishing either Sensor_msgs/LaserScan or Sensor_msgs/PointCloud messages over ROS.

⇒ The navigation stack requires that odometry information be published using tf and the nav_msgs/Odometry message.

⇒ The navigation stack assumes that it can send velocity commands using a geometry_msgs/Twist message assumed to be in the base coordinate frame of the robot on the "cmd_vel" topic.

⇒ map_server is optional for navigation stack.

4.

Publishing Odometry Information over ROS

⇒ The navigation stack uses tf to determine the robot's location in the world and relate sensor data to a static map.

↳ Tf does not provide any information about the velocity of the robot.

⇒ Because of this, the navigation stack requires that any odometry source publish both a transform and a nav_msgs/Odometry message over ROS that contains velocity information.

2. The nav_msgs/Odometry Message

=> The nav_msgs/Odometry message stores an estimate of the position and velocity of a robot in free space.

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

CHAPTER 29

Running ROS across multiple machine

Running ROS across multiple machine

↳ It explains the use of ROS_MASTER_URI to configure multiple machine to use a single master.

* Overview

⇒ ROS is designed with distributed computing in mind.

⇒ A well-written node makes no assumptions about where in the network it runs, allowing computation to be relocated at runtime to match the available resources.

↳ Exception is driver node that communicate with a piece of hardware.

⇒ Deploying a ROS system across multiple machines :-

- You 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 complex, bi-directional connectivity between all pairs of machine, on all ports.
- Each machine must advertise itself by a name that all other machines can resolve.

In raspberry Pi

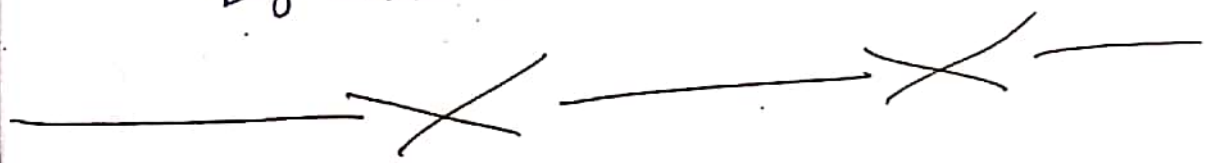
ssh adityashiwastava35@coffee_bot
export ROS_MASTER_URI = http://
coffeebot:11311

=== run any node ===

On Laptop

ssh aditya@laptop
export ROS_MASTER_URI = http://coffeebot:11311

=== run any node or subscribe ===
to any topic that is hosted
by coffeebot.



ROS/Network Setup

⇒ ROS has certain requirements of the network configuration:

- ⇒ There must be complete bi-directional connectivity between all pairs of machine, on all ports.
- ⇒ Each machine must advertise itself by a name that all other machine can resolve.

⇒ In the following sections, we'll assume that you want to run a ROS system on two machines, with the following hostnames and IP address.

- marvin.example.com : 192.168.1.1
- hal.example.com : 192.168.1.2

* Name resolution

⇒ When a ROS node advertises a topic, it provides a hostname:port combination (a URI) that other nodes will contact when they want to subscribe to that topic.

URI ⇒ A Uniform resource Identifier is a string of characters that unambiguously identifies a particular resource.

⇒ Most common form of URI is the Uniform Resource Locator (URL)

⊕ Setting a name explicitly

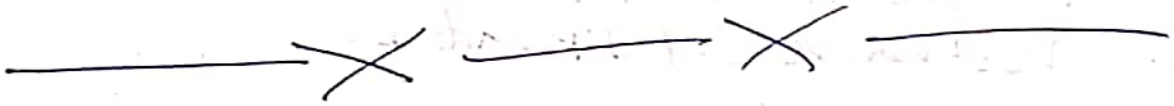
⇒ If a machine reports a hostname that is not addressable by other machines, then you need to set either the ROS_IP or ROS_HOSTNAME environment variable.

In /etc/hosts

↓

192.168.43.18 Coffeebot

192.168.43.21 laptop



CHAPTER 30

RPLiDAR

RP LiDAR

* Overview

⇒ This package provides basic device handling for 2D Laser Scanner RPLIDAR A3/A2 & A1.

⇒ Suitable for indoor robotic slam application.

- 360 degree scan field
- 5.5hz/10hz rotating frequency
  - ↳ User can customize the scanning frequency from 2hz to 10hz freely

⇒ The driver published device-dependent Sensor_msgs/LaserScan data.

* ROS Nodes

① rplidarNode

⇒ driver for RPLIDAR

⇒ reads raw scan result using RPLIDAR's SDK and convert to ROS.

Published Topics

↳ scan (Sensor_msgs/LaserScan)

Services

↳ stop_motor (std_srvs/Empty)

⇒ Call the service to stop the motor of RPLIDAR.

↳ start_motor (std_srvs/Empty)

⇒ Call the service to start the motor of rplidar.

Parameters

↳ serial_port (string, default: /dev/ttyUSB0)

↳ serial_baudrate (int, default: 115200)

↳ frame_id (string, default: laser)

↳ inverted (bool, default: false)

⇒ indicated whether the Lidar is mounted inverted.

↳ angle_compensate (bool, default: true)

⇒ indicate whether the driver need do angle comp.

↳ scan_mode (string, std::string())

⇒ The scan mode of lidar.

⇒ Before running RPLIDAR driver set the permissions.

```
sudo chmod 666 /dev/ttyUSB0
```

CHAPTER 31

Gmapping

Gmapping

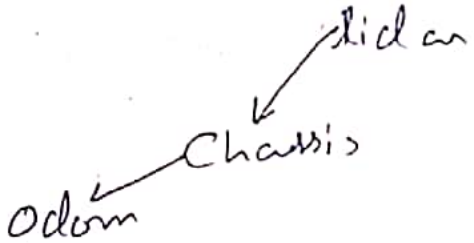
→ Using this you can create 2D occupancy grid map from laser and pose data collected by a mobile robot.

Hardware requirement

→ Horizontally mounted fixed laser range finder.
  → It will publish laser scan data.

→ Odometry data

→ tf ⟹



⟹ It builds a map (nav_msgs/OccupancyGrid)

Subscribed topic

① tf (tf/tfMessage)

② scan (sensor_msgs/LaserScan)

Published topic

① map_metadata (nav_msgs/MapMetaData)

② map (nav_msgs/OccupancyGrid)

③ ~entropy (std_msgs/Float64)

Service

① dynamic_map (nav_msgs/GetMap)

⇒ To save a map :-

```
rosrun map_server map_saver
  -f <map_name>
```

d)

CHAPTER 32

Map Server

Map_Server

⇒ Offers map data as a ROS Service.

* Map format

⇒ Map is stored in a pair of files.

- YAML file describes the map meta-data, and names the image file.
- The image file encodes the occupancy data.
  - White ⇒ free
  - black ⇒ occupied
  - Pixels color in between ⇒ Unknown

rosrun map_server map_server map_name.yaml

CHAPTER 33

AMCL

AMCL

⇒ Amcl is a probabilistic localization system for a robot moving in 2D.

⇒ It implements the adaptive (KLD-sampling) Monte Carlo localization approach

⇒ amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates.

⇒ On startup, amcl initializes its particle filter according to the parameters provided, if no parameters are set, the initial filter state will be a moderately sized particle cloud centered about (0,0,0).

★ Subscribed topic

scan ⟶ laser scan

tf ⟶ Transforms

map ⟶ When the use_map_topic Parameter is Set

initialpose ⟶ Mean and covariance with which to re-initialize the particle filter.

★ Published topics

amcl_pose ⟶ Robot's estimated pose in the map, with covariance.

particlecloud
↳ The set of pose estimates being maintained by the filter.

tf
↳ Publishes transform from odom to map.

★ Services

global_localization
↳ Initiate global localization, wherein all particles are dispersed randomly through the free space in the map.

request_nomotion_update
↳ Service to manually perform update and publish updated Particles.

set_map
↳ Service to manually set a new map and pose.

★ Service called

static_map
↳ amcl calls this service to retrieve the map that is used for laser based localization.