ROS: Robot Operating System

for
AGV KGP

by
Jit Ray Chowdhury

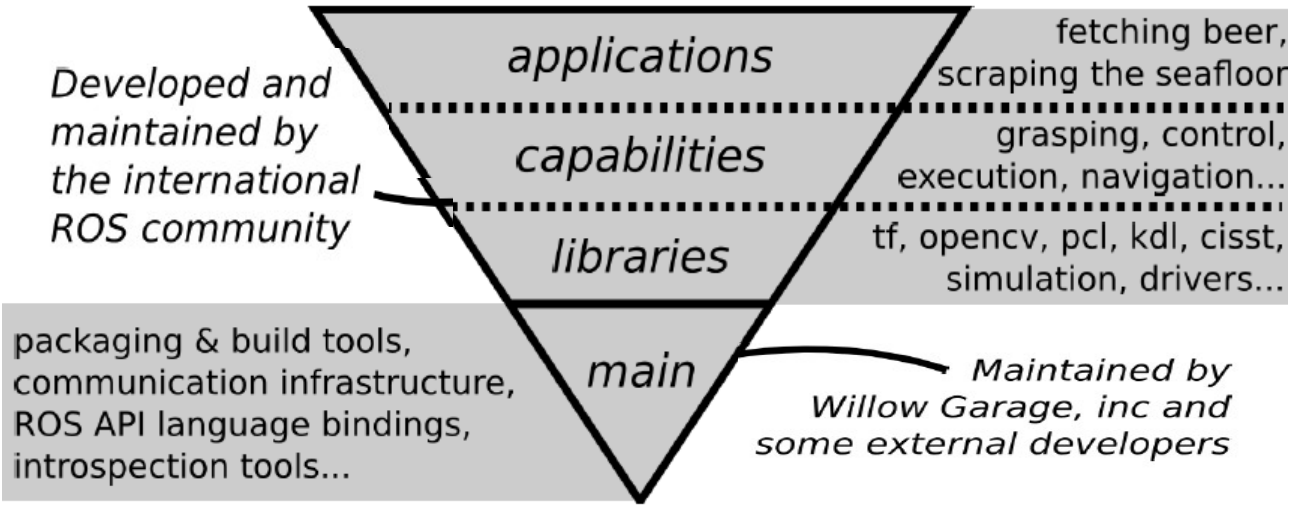
Table of Contents

What is ROS?

- A “meta” operating system for robots
- A collection of packaging, software building tools
- An architecture for distributed inter-process/inter-machine communication and configuration
- Development tools for system runtime and data analysis
- A language-independent architecture (c++, python, lisp, java, and more)

What is ROS not

- An actual operating system
- A programming language
- A programming environment / IDE
- A hard real-time architecture



ROS Distributions

Collection of stacks and roscore …

- Groovy Galapagos (Oct 2012)
- ROS Fuerte Turtle, released April 23, 2012 (Recommended)
- ROS Electric Emys, released August 30, 2011
- ROS Diamondback, released March 2, 2011
- ROS C Turtle, released August 2, 2010
- ROS Box Turtle, released March 2, 2010

ROS and its components

ROS Core

ROS Master

- A centralized XML-RPC server
- Negotiates communication connections
- Registers and looks up names for ROS graph resources

Parameter Server

- Stores persistent configuration parameters and other arbitrary data

rosout

- Essentially a network-based stdout for human-readable messages

ROS Stacks & Packages

ROS code is grouped at two different levels:

- Packages
    - A named collection of software that is built and treated as an atomic dependency in the ROS build system.
- Stacks
    - A named collection of packages for distribution.

Packages

- A folder that contains your code, build files, launch files, etc.
- Can contain any number of nodes
- 'manifest.xml' – lists the ROS dependencies & system deps
- Should only contain code that is related
- ex. laser pipeline, motor controllers, localization, SLAM, forward kinematics, Hokuyo driver...

Nodes

- is a process that performs some function.
- nodes communicate with each other using topics & services.
- nodes are assigned unique names
- odes are intended to be modular and 'operate on the fine-grained scale'

Build System

- Need to specify in 'Cmakelists.txt' how to build the source code of a package
- 'rosmake': compile pkg + deps
- Can download system dependencies if not installed
- Compile multiple pkgs in parallel. ROS resolves deps first. 'ROS_PARALLEL_JOBS' = # of cores

Command Line Tools

Command	Description
roscd	Change directory to specified ros-package
rosls	List contents of a ros-package
rosmake	Build all of the ros packages that a package depends on ●'--pre-clean': first run 'make clean' in each package then run 'make' ●'--rosdep-install': install system dependencies first then compile all
roslaunch	Launch a '.launch' file (looks in 'launch' directory for file)
roscreate-pkg	roscreate-pkg Create a ros-package ●State the name & dependencies ●Automatically generates the directory, manifest.xml, Makefile, etc. ●Can always change anything later!
rosdep	State dependencies of a package, Find out what depends on a specific package, Capable of output in tree format
roscp	Copy files from one package to another
rosed	Bring up your default text editor and edit file ”ros_package filename.txt”
rostest	Execute a regression test file

Ways to communicate

- Topic (pub sub)
    - Asynchronous "stream-like" communication
    - Strongly-typed (ROS .msg spec)
    - Callback function is multi-threaded
    - Not appropriate for request/reply interaction
    - Many-to-many
    - ex. 'base_scan' is a publisher that publishes laser scans at 10 hz. Global planner, Controller & localization nodes subscribe to base_scan.
- Service (higher priority)
    - Synchronous "function-call-like" communication
    - Strongly-typed (ROS .srv spec)
    - one-to-one
    - Can have one or more clients
    - No topic callbacks are issued during service call (service request is blocking)
    - ex. request a motion plan
- Actions
    - Built on top of topics
    - Long running processes

- Cancellation
- http://www.ros.org/wiki/actionlib
    - **Goal:** For controlling the tilting laser scanner, the goal would contain the scan parameters (min angle, max angle, speed, etc).
    - **Feedback:** For controlling the tilting laser scanner, this might be the time left until the scan completes.
    - **Result:** For controlling the tilting laser scanner, the result might contain a point cloud generated from the requested scan.

Messages

- nodes communicate by passing around messages
- a message is a data structure with typed fields
- many standard messages already exist, new messages can be defined with a simple text file
- a message can be comprised of other messages
- ROS generates a data structure for new message that contains many standard stl type of functions (size(), resize(),etc.)

Params

- a parameter server that stores parameter strings & value pairs which are normally passed as input to a program
- some params can be viewed by other nodes
- great way to pass around a name of a topic or other info multiple nodes might need to know
- can put XML & YAML files onto server
- ex. ”shoulder_pan_max_vel” → '0.7' (double)
- ex. ”camera_resolution_640_480” → 'true' (bool)
- ex. ”type_of_planner” → ”ARA” (string)

Launch file

- A launch file is a convenient way to bringup many different **nodes** at once
- Written in XML
- Asynchronous execution
- Can put parameters on server
- Hierarchically compose collections of other launch files
- Automatically re-spawn nodes if they crash
- Change node names, namespaces, topics, and other resource names without recompiling
- Easily distribute nodes across multiple machines

Debugging

- rxgraph: displays a visualization of the ROS graph – the ROS nodes that are currently running and the topics that connect them
- rxplot: plot data from one or more ROS topic fields that are currently being published.
- rxconsole: brings up a viewer that displays any messages being published to 'rosout'
    - can display the filename & line number of a message
    - useful for debugging code you are unfamiliar with
- rostopic
- roswtf
- rosnode
- rosservice
- rosmsg: get field names and field types of a message
- rossrv: get the field names and field types of a service request/reply message

ROS graph resources

- nodes
    - processes
    - produce and consume data
- parameters
    - persistent data storage
    - configuration, initialization settings
    - stored on parameter server
- topics
    - Asynchronous many-to-many communication streams.
- services
    - Synchronous one-to-many network-based functions.

Rosout

ROS provides mechanisms in all languages for specifying different levels

of human-readable log messages.

The five default levels are:

1. ROS_FATAL(...)
2. ROS_ERROR(...)
3. ROS_WARN(...)
4. ROS_INFO(...)

5. ROS_DEBUG(...)

control by **rxconsole**

Simulation: Stage

2d simulator

Simulation: Gazebo

3d simulator

Visualizers: rviz

- capable of displaying all 'visualizable' messages without extra coding
- 'nav_view' is a 2D version of rviz

ROS Play/Record

- can record any information passed over ROS to a 'bag' file
- the file can be played back later
- ex. log sensor data for later analysis
- ex. great for debugging hard to recreate situations

http://www.ros.org/wiki/ROS/Tutorials/Recording and playing back data

glc-record

- record gazebo & rviz windows at the same time to create a multi-window video
- can easily record many OpenGL apps simultaneously

ROS in more details

ROS Meta-Filesystem

The minimal representation of a ROS package is a directory in the
$ROS PACKAGE PATH which contains a single file:

- manifest.xml
    - Contains package metadata (author, license, url, etc)
    - Specifies system and package dependencies
    - Specifies language-specific export flags
- CMakeLists.txt: contains ROS build rules (executables, libraries, custom build flags, etc)
- Makefile: just a proxy to build this package
- Create package with roscreate-pkg

```
$ roscreate-pkg foo roscpp std_msgs
```

- Build package with rosmake

```
$ rosmake foo
```

ROSCPP

- Initialization with ros::init:
  - register at core
  - set up remappings
  - set up networking
- ros::NodeHandle as interface to topics, services and parameters
- ros::NodeHandle::subscribe, ros::NodeHandle::advertise for topics
- ros::spin and ros::spinOnce to process ROS messages
- Use boost::bind to use member functions as callbacks:
  - boost::bind (Listener::laserCb, this , -1) ;

Messages Structure

- defined in package-name/msg/*.msg files, sent over topics
- basic data types:
  - int{8,16,32,64}
  - float{32,64}
  - string
  - time
  - duration
  - array[]
- Example: Point.msg

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

Services

- Defined in package-name/srv/*.srv.
- Definition similar to message files, Request message + response message.
- Example: beginner tutorials/AddTwoInts

```
int64 a
int64 b
---
```

int64 sum

Getting started with ROS

Create a simple node

- create a new package
- write your code (usually as a class)
- create a main function that instantiates class
- list the dependencies
- describe how it should be built
- build it
- create a launch file
- use rviz to inspect it's working correctly

Why contain your node's functionality in a class?

- you will have many shared variables that you don't want to pass around as parameters between functions (publishers,subscribers,transforms, node handles)
- have a main function that instantiates the class, and then calls ros::spin() // wait for shutdown

Using Messages

- use 'rosmsg show …' to remind yourself of field names and types (or go to ros.org)
- remember to include the message header file with the correct case
    - <mapping_msgs/CollisionMap.h>

Simple Publisher (C++)

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

/**
 * This tutorial demonstrates simple sending of messages over the ROS
system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can
perform
   * any ROS arguments and name remapping that were provided at the
command line. For programmatic
```

```
   * remappings you can use a different version of init() which takes
remappings
   * directly, but for most command-line programs, passing argc and argv
is the easiest
   * way to do it.  The third argument to init() is the name of the
node.
   *
   * You must call one of the versions of ros::init() before using any
other
   * part of the ROS system.
   */
  ros::init(argc, argv, "talker");

  /**
   * NodeHandle is the main access point to communications with the ROS
system.
   * The first NodeHandle constructed will fully initialize this node,
and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * 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 and who
   * is subscribing. After this advertise() call is made, the 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 messages on that topic through a call to publish().  Once
   * all copies of the returned Publisher object are destroyed, the
topic
   * will be automatically unadvertised.
   *
   * The second parameter to advertise() is the size of the message
queue
   * used for publishing messages.  If messages are published more
quickly
   * than we can send them, the number here specifies how many messages
to
   * buffer up before throwing some away.
   */
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter",
1000);

  ros::Rate loop_rate(10);

  /**
   * A count of how many messages we have sent. This is used to create
   * a unique string for each message.
   */
  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then
publish it.
```

```
     */
    std_msgs::String msg;

    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

    ROS_INFO("%s", msg.data.c_str());

    /**
     * The publish() function is how you send messages. The parameter
     * is the message object. The type of this object must agree with
the type
     * given as a template parameter to the advertise<>() call, as was
done
     * in the constructor above.
     */
    chatter_pub.publish(msg);

    ros::spinOnce();

    loop_rate.sleep();
    ++count;
  }


  return 0;
}
```

Simple Subscriber (C++)

```
#include "ros/ros.h"
#include "std_msgs/String.h"

/**
 * This tutorial demonstrates simple receipt of messages over the ROS
system.
 */
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can
perform
   * any ROS arguments and name remapping that were provided at the
command line. For programmatic
   * remappings you can use a different version of init() which takes
remappings
   * directly, but for most command-line programs, passing argc and argv
is the easiest
   * way to do it.  The third argument to init() is the name of the
node.
   *
   * You must call one of the versions of ros::init() before using any
other
```

```
   * part of the ROS system.
   */
  ros::init(argc, argv, "listener");

  /**
   * NodeHandle is the main access point to communications with the ROS
system.
   * The first NodeHandle constructed will fully initialize this node,
and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The subscribe() call is how you tell ROS that you want to receive
messages
   * on a given topic.  This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing.  Messages are passed to a callback function, here
   * called chatterCallback.  subscribe() returns a Subscriber object
that you
   * must hold on to until you want to unsubscribe.  When all copies of
the Subscriber
   * object go out of scope, this callback will automatically be
unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is the size of
the message
   * queue.  If messages are arriving faster than they are being
processed, this
   * is the number of messages that will be buffered up before beginning
to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  /**
   * ros::spin() will enter a loop, pumping callbacks.  With this
version, all
   * callbacks will be called from within this thread (the main one).
ros::spin()
   * will exit when Ctrl-C is pressed, or the node is shutdown by the
master.
   */
  ros::spin();

  return 0;
}
```

Service Server

src/add_two_ints_server.cpp

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

bool add(beginner_tutorials::AddTwoInts::Request  &req,
```

```
        beginner_tutorials::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);
  return true;
}

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

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

  return 0;
}
```

Service Client

src/add_two_ints_client.cpp

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

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 =
n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::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;
}
```

Action Definitions

- Similar to messages and services.
- Definition: Request + result + feedback
- Defined in ros-package/action/*.action
- Generated by CMake macro genaction().
- Example: actionlib tutorials/Fibonacci.action

```
#goal definition
int32 order
---
#result definition
int32[] sequence
---
#feedback
int32[] sequence

roscd learning_actionlib
$ rosrun actionlib_msgs genaction.py -o msg
action/Fibonacci.action
```

more at
http://www.ros.org/wiki/actionlib_tutorials/Tutorials/SimpleActionServer(ExecuteCallbackMethod)

Example

Launch file example

```
<launch>
        <!-- load empty world -->
        <include file="$(find pr2_gazebo)/pr2_empty_world.launch"/>
        <!-- load planning -->
        <include file="$(find sbpl_arm_planner)/launch/sbpl_planning_right_arm.launch"/>
        <!-- load common nodes for motion planning tests -->
        <include file="$(find
        arm_navigation_tests)/tests/motion_planers/common/motion_planning_common_right_arm.
        launch"/>


        <!-- tuck left arm-->
        <node pkg="pr2_experimental_controllers" type="tuckarm.py" args="l" output="screen" >
                <param name="planner_service_name" value="/sbpl_planning/plan_path"/>
                <param name="planner_id" value="435"/>
        </node>


        <node name="my_node" pkg="foo" type="bar">
                <remap f rom="/base_laser/scan " to="scan " />
                <rosparam>
                        usefoo : True
                        frameid : base_laser
                </rosparam>
        </node>
</launch>
```

Stage

http://www.ros.org/wiki/stage/Tutorials/SimulatingOneRobot

roscore

```
rosmake stage
```

rosrun stage stageros `rospack find stage`/world/willow-erratic.world

```
svn co https://code.ros.org/svn/wg-ros-
pkg/branches/trunk_cturtle/sandbox/teleop_base teleop_base

rosmake teleop_base
```

```
rosrun teleop_base teleop_base_keyboard base_controller/command:=cmd_vel

rosmake rviz

roscd stage

rosrun rviz rviz -d `rospack find stage`/rviz/stage.vcg
```

more at:

http://www.ros.org/wiki/stage/Tutorials/IntroductiontoStageControllers

Initializing Gazebo Simulation

roslaunch gazebo_worlds empty_world.launch

rosservice list gazebo

roscd gazebo_worlds/

rosrun gazebo spawn_model -file objects/desk1.model -gazebo -model desk1 -x 0

rosservice call gazebo/get_world_properties

rosservice call gazebo/get_model_properties table_model

rostopic echo -n 1 /gazebo/model_states

rostopic echo -n 1 /gazebo/link_states

Applying Forces

rosrun gazebo spawn_model -file `rospack find gazebo_worlds`/objects/000.580.67.model -gazebo -model cup -z 1

Writing a Package in ROS (C++)

Refer page 16 of ROS_Tutorial.pdf

Communication with a P3DX robot by reading a topic

Refer page 18 of ROS_Tutorial.pdf

- roscreate-pkg p3dxReader std_msgs gazebo nav_msgs roscpp
- create and edit `src/reader.cpp`

```
#include "ros/ros.h"
#include "nav_msgs/Odometry.h"
void callback(const nav_msgs::Odometry::ConstPtr& str)
{
	printf("P3DXReader-> Reading Message %f,%f\n",
```

```
	str->pose.pose.position.x, str->pose.pose.position.y);
}
int main(int argc, char **argv)
{
	ros::init(argc, argv, "reader");
	ros::NodeHandle n;
	ros::Subscriber sub = n.subscribe("/erratic_odometry/odom", 1000,
	callback);
	printf("P3DX Reader initialized\n");
	ros::spin();
	return 0;
}
```

Add to CmakeLists.txt

- rosbuild_add_executable(reader src/reader.cpp)

Can be used with Gazebo

- roslaunch gazeroslaunch gazebo_worlds empty_world.launch
- roslaunch p3dx.launch
- rostopic pub -1 /cmd_vel geometry_msgs/Twist '{linear: {x: 1.0, y: 0.0, z: 0.0}, angular: { x: 0.0, y: 0.0, z: 1.0} }'

Writing a ROS publisher in C++

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
int main(int argc, char **argv)
{
	ros::init(argc, argv, "publisher");
	ros::NodeHandle node;
	ros::Publisher p3dxCmdPub = node.advertise<geometry_msgs::Twist>("cmd_vel", 1000);
	ros::Rate loop_rate(10);
	int count = 0;
	while (ros::ok())
	{
		geometry_msgs::Twist newSpeed;
		newSpeed.linear.x=1.0;
		newSpeed.linear.y=0.0;
		newSpeed.linear.z=0.0;
		newSpeed.angular.x=0.0;
		newSpeed.angular.y=0.0;
		newSpeed.angular.z=1.0;
		p3dxCmdPub.publish(newSpeed);
		ros::spinOnce();
		loop_rate.sleep();
		++count;
	}
	return 0;
}
```

Add to CMakeLists.txt

- rosbuild_add_executable(publisher src/publisher.cpp)

Writing a Simple Image Publisher

http://www.ros.org/wiki/image_transport/Tutorials/PublishingImages

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv/cvwimage.h>
#include <opencv/highgui.h>
#include <cv_bridge/CvBridge.h>

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

```
  ros::init(argc, argv, "image_publisher");
  ros::NodeHandle nh;
  image_transport::ImageTransport it(nh);
  image_transport::Publisher pub = it.advertise("camera/image", 1);

  cv::WImageBuffer3_b image( cvLoadImage(argv[1],
CV_LOAD_IMAGE_COLOR) );
  sensor_msgs::ImagePtr msg =
sensor_msgs::CvBridge::cvToImgMsg(image.Ipl(), "bgr8");

  ros::Rate loop_rate(5);
  while (nh.ok()) {
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

Our AGV Bot

Install in Ubuntu 12.04 (Precise)

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

wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update


sudo apt-get install ros-fuerte-desktop-full
      460 MB of archives, 1,540 MB of additional disk space
or
sudo apt-get install ros-fuerte-desktop
      275 MB of archives, 965 MB of additional disk space
or
sudo apt-get install ros-fuerte-ros-comm
      11.6 MB of archives, 54.4 MB of additional disk space


sudo apt-get install ros-fuerte-rx
sudo apt-get install python-rosinstall python-rosdep

gedit ~/.bashrc
source /opt/ros/fuerte/setup.bash
export ROS_WORKSPACE=~/fuerte_workspace/
export ROS_PACKAGE_PATH+=:~/fuerte_workspace/sandbox/

mkdir -p ~/fuerte_workspace/sandbox/


sudo rosdep init
rosdep update



roscd
cd sandbox
roscreate-pkg beginner_tutorials std_msgs rospy roscpp
rospack profile
rospack find beginner_tutorials

gedit beginner_tutorials/src/helloROS.cpp
#include <iostream>
using namespace std;
int main()
{
      cout<<"Hello";
      return 0;
}

gedit  beginner_tutorials/CMakeLists.txt
rosbuild_add_executable(hello src/helloROS.cpp)

rosmake beginner_tutorials
rosrun beginner_tutorials hello
```

The Plan

Param server for setting config like serial port, baudrate for various communication devices
stack Eklavya
modules as packages
packages can contain multiple nodes, msg …

Miscellaneous

The TF Library (Transform Frame)

sudo apt-get install ssh ros-fuerte-turtlebot* ros-fuerte-viz

rosmsg list
rosmsg show sensor_msgs/Image

Eclipse IDE

make eclipse-project

cmake -G"Eclipse CDT4 - Unix Makefiles"

http://www.ros.org/wiki/IDEs

Links

Learn

http://www.ros.org/wiki/Courses

http://www.ros.org/wiki/ROS/Tutorials
https://alliance.seas.upenn.edu/~meam620/wiki/index.php?n=Roslab.ROSTutorials

https://wiki.nps.edu/display/~thchung/ROS+--+Gazebo+Simulator
http://www.willowgarage.com/blog/2009/12/01/ros-tutorials-turtles
http://mrl.isr.uc.pt/events/iros2012tutorial/
http://www.ros.org/wiki/tf/Tutorials

http://answers.ros.org/question/12599/ros-beginning/#18583
http://www.ros.org/wiki/turtlebot_follower/Tutorials/Demo
http://www.ros.org/wiki/Robots/TurtleBot
http://www.ros.org/wiki/APIs

http://www.ros.org/wiki/simulator_gazebo/Tutorials/StartingGazebo
http://www.ros.org/wiki/AllTutorials

http://www.ros.org/wiki/navigation/Tutorials/RobotSetup
http://www.willowgarage.com/blog/2012/01/16/capturing-accurate-camera-poses
http://www.pirobot.org/blog/0016/

http://www.ros.org/wiki/IDEs

Robotics News

http://nootrix.com/articles/
http://www.ros.org/news/robots/