Control of Mobile Robots Using ActionLib

Higor Barbosa Santos, Marco Antônio Simões Teixeira, André Schneider de Oliveira, Lúcia Valéria Ramos de Arruda and Flávio Neves Jr.

Abstract Mobile robots are very complex systems and involve the integration of various structures (mechanical, software and electronics). The robot control system must integrate these structures so that it can perform its tasks properly. Mobile robots use control strategies for many reasons, like velocity control of wheels, position control and path tracking. These controllers require the use of preemptive structures. Therefore, this tutorial chapter aims to clarify the design of controllers for mobile robots based on ROS ActionLib. Each controller is designed in an individual ROS node to allow parallel processing by operating system. To exemplify the controller design using ActionLib, this chapter will demonstrate the implementation of two different types of controllers (PID and Fuzzy) for position control of a servo motor. These controllers will be available on GitHub. Also, a case study of scheduled fuzzy controllers based on ROS ActionLib for a magnetic climber robot used in the inspection of spherical tanks will be shown.

Keywords ROS · Mobile robots · Control · ActionLib

1 Introduction

Mobile robots have great versatility because they're free to run around their application environment. However, this is only possible because this kind of robot carries a great variety of exteroceptive and interoceptive sensors to measure its motion and

H.B. Santos (✉) · M.A.S. Teixeira · A.S. de Oliveira · L.V.R. de Arruda · F. Neves Jr.
Federal University of Technology—Parana, Av. Sete de Setembro, 3165 Curitiba, Brazil
e-mail: higorsantos@alunos.utfpr.edu.br

M.A.S. Teixeira
e-mail: marcoteixeira@alunos.utfpr.edu.br

A.S. de Oliveira
e-mail: andreoliveira@utfpr.edu.br

L.V.R. de Arruda
e-mail: lvrarruda@utfpr.edu.br

F. Neves Jr.
e-mail: neves@utfpr.edu.br

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 707, DOI 10.1007/978-3-319-54927-9_5

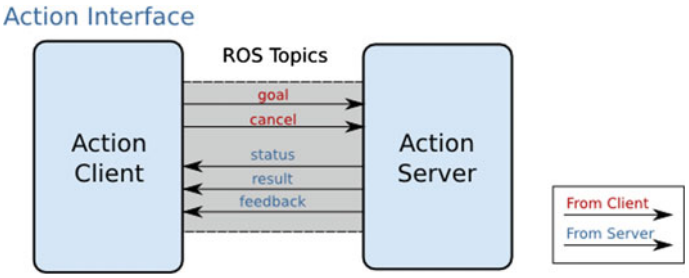

Fig. 1 Interface of ActionLib. *Source* [1]

interact with environment around it. Several of these information are used to robot's odometry or environment mapping. Thus, these signals are the robot's sense about its motion and its way to correct it.

Robot control is a complex and essential task which must be performed during all navigation. Several kinds of controllers can be applied (like proportional-integral-derivative, predictive, robust, adaptive and fuzzy). The implementation of the robot control is very similar and can be developed with the use of ROS Action Protocol by ActionLib. ActionLib provides a structure to create servers that execute long-running tasks and interacts with clients by specific messages [1].

The proposed chapter aims to explain how to create ROS controllers using the ActionLib structure. This chapter is structured in five sections.

In the first section, we will carefully discuss the ActionLib structure. This section introduces the development of preemptive tasks with ROS. The ActionLib works with three main messages, as can be seen in Fig. 1. Goal message is the desired value for controller, like its target or objective. Feedback message is the measure of controlled variable which usually is updated by means of a robot sensor. Result message is a flag that indicates when the controller reaches its goal.

The second section will demonstrate the initial requirements for the creation of a controller using ActionLib package.

The third section will present an implementation of a classic Proportional-Derivative-Integrative (PID) control strategy with the use of ActionLib. This section aims to introduce a simple (but powerful) control structure that can be applied to many different purposes, like position control, velocity control, flight control, adhesion control and among others. It'll be shown how to create your own package, set the action message, structure the server/client code, compile the created package and, finally, show the experimental results of the PID controller.

In the fourth section will be present a design of a fuzzy controller. The controller is implemented using ActionLib and an open-source fuzzy library. The fuzzy logic enables the implementation of a control without the knowledge of the system dynamics model.

Finally, last section will show a study case of an ActionLib based control for the second-generation of a climbing robot with four steerable magnetic wheels [2], called

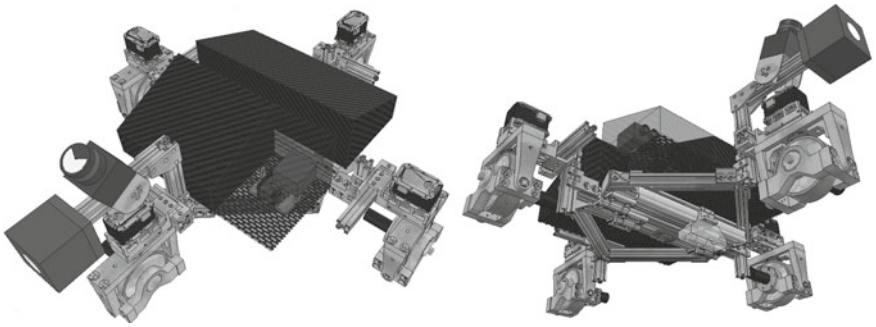

Fig. 2 Autonomous inspection robot (AIR-2)

as Autonomous Inspection Robot 2^{nd} generation (AIR-2), as shown in Fig. 2. AIR-2 is a robot fully compatible with ROS and it has a mechanical structure designed to provide high mobility when climbs on industrial storage tanks. The scheduled fuzzy controllers were designed to manage the speed of AIR-2.

2 ActionLib

Mobile robots are very complex, they have many sensors and actuators that help them get around and locate in an unknown environment. The control of the robot isn't an easy task and it should have a parallel processing. The mobile robot must handle several tasks at same time, so the preemption is an important feature to robot control. Often, the robot control is multivariable, that's means multiple-input multiple-output systems (MIMO). Therefore, developing an algorithm with these characteristics is hard.

Robot control covers various functions of a robot. For example, obstacle avoidance is very important for the autonomous mobile robots. Therefore, [3] proposed a fuzzy intelligent obstacle avoidance controller for a wheeled mobile robot. Balancing robot is another relevant aspect in robotics, said that, [4] designed a cascaded PID controller for movement control of a two wheel robot. On the other hand, [5] presented an adhesion force control for a magnetic climbing robot used in the inspection of storage tanks. Therefore, the robot control is essential to ensure its operation, either navigation or obstacle avoidance.

The ROS has libraries that help in the implementation of control, like *ros_control*. Other library is ActionLib that enables to create servers to execute long-running tasks and clients that interact with servers. Given these features, the development of a controller using this library becomes easy. On the other hand, *ros_control* package is hard to be used, it presents a control structure that requires many configurations for implementation of a specific controller, for example, a fuzzy controller.

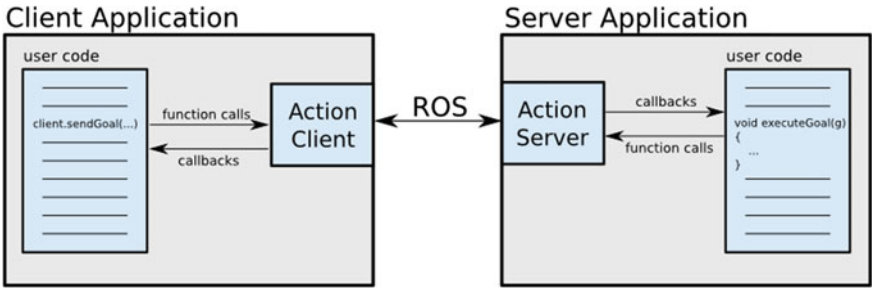

Fig. 3 Client-server interaction. *Source* [1]

The Actionlib provides a simple application for client sends goals and server executes goals. The server executes long-running goals that can be preempted. Client-server interaction using ROS Action Protocol is shown in Fig. 3.

The client-server interaction in ActionLib is provided by messages that are displayed in Fig. 1. The messages are:

- **goal**: client sends goal to the server;
- **cancel**: client sends the cancellation of the goal to the server;
- **status**: server notifies the status of the goal for the client;
- **feedback**: server sends goal information to the client;
- **result**: server notifies the client when the goal was achieved.

Thus, the ActionLib package is a powerful tool for the design of controllers in robotics. In the next section, the initial configuration of ROS Workspace will be presented for the use of ActionLib package.

3 ROS Workspace Configuration

For implementation of the controller it's necessary that ROS Indigo is properly installed. It's available at:

http://wiki.ros.org/indigo/Installation/Ubuntu

The next step is the ROS Workspace configuration. If it isn't configured on your machine, it'll be necessary create it:

```
$ mkdir -p /home/user/catkin_ws/src
$ cd /home/user/catkin_ws/src
$ catkin_init_workspace
```

The *catkin_init_workspace* command sets the *catkin_ws* folder as your workspace. After, you must build the workspace. For this, you need to navigate to your workspace folder and then type the command *catkin_make*, as shown below.

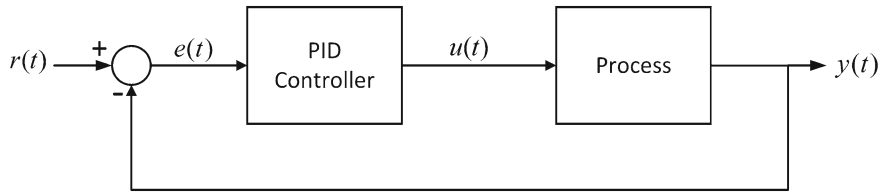

Fig. 4 PID control

```
$ cd /home/user/catkin_ws/
$ catkin_make
```

To add the workspace to your ROS environment, you need to source the generated setup file:

```
$ source /home/user/catkin_ws/devel/setup.bash
```

After workspace configuration, we can start creating a PID controller using ActionLib, which it'll be shown in the next section.

4 Creating a PID Controller Using ActionLib

There're various types of algorithm used to robot control. But the PID control is more used, due to its good performance for linearized systems and easy implementation. The PID controller is a control loop feedback widely used in various applications, in Fig. 4 can be seen its diagram. The Eq. 1 shows the PID equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

$$e(t) = r(t) - y(t) \tag{2}$$

where $u(t)$ is output, K_p is proportional gain, $e(t)$ is error (difference between setpoint $r(t)$ and process output $y(t)$, as shown in Eq. 2), K_i is integral gain and K_d is derivative gain. The controller calculates the error and by adjusting the gains (K_p, K_i and K_d), the PID seeks to minimize it.

In this section, it'll be shown the PID control implementation using ActionLib. The PID will control the angle of a servo motor. The servo motor was simulated in robot simulator V-REP. The V-REP is simulator based on distributed control architecture, it allows the modeling of robotic systems similar to the reality [6].

The controller has been implemented in accordance with Fig. 4, in which the setpoint is the desired angle (goal) and the feedback is provided by encoder servo.

The PID controller is available on GitHub and can be installed on your workspace:

```
$ source /opt/ros/indigo/setup.bash
$ cd /home/user/catkin_ws/src
$ git clone https://github.com/air-lasca/tutorial_controller
$ cd ..
$ catkin_make
$ source /home/user/catkin_ws/devel/setup.bash
```

The following will be shown its creation step-by-step.

4.1 Steps to Create the Controller

1st **step: Creating the ActionLib package** Once the workspace is created and configured, let's create package using ActionLib:

```
$ cd /home/user/catkin_ws/src/
$ catkin_create_pkg tutorial_controller actionlib
    message_generation roscpp rospy std_msgs actionlib_msgs
```

The *catkin_create_package* command creates a package named *tutorial_controller* which depends on *actionlib*, *message_generation*, *roscpp*, *rospy*, *std_msgs* and *actionlib_msgs*. Posteriorly, if you need other dependencies just add them in *CMakelist.txt*. This will be detailed in fifth step.

After creating the package, we need to define the message that is sent between the server and client.

2nd **step: Creating the action messages** Continuing steps to create the controller, you must set the action messages. The action file has three parts: goal, result and feedback. Each section of action file is separated by 3 hyphens (- - -).

The goal message is the setpoint of controller, it is sent from the client to the server. Yet, the result message is sent from the server to the client, it tells us when the server completed the goal. It would be a flag to indicate that the controller has reached the goal, but for the control has no purpose. While, the feedback message is sent by the server to inform the goal of incremental progress for the client. Feedback would be the information from the sensor used in control.

Then, to create the action messages, you must create a folder called *action* in your package.

```
$ cd /home/user/catkin_ws/src/tutorial_controller
$ mkdir action
```

After creating the folder, you must create an .action file (*Tutorial.action*) in action's folder of your package. The first letter of the action name should be uppercase. This information is placed in the .action file:

```
#Define the goal
float64 position
---
```

```
#Define the result
bool ok
---
#Define a feedback message
float64 position
```

The goal and feedback are defined as *float64*. The goal will receive the desired position of a servo motor and feedback will be used to send the information acquired from the encoder servo. The result is defined as *bool*, but it won't be used in control. This action file will be used in the controllers examples shown in this chapter.

The action messages are generated automatically from the .action file.

3^{rd} **step: Create the action client** In src folder of your package, create *ControllerClient.cpp*, it'll be client of ActionLib. Firstly, we'll include the necessary libraries of ROS, action message and action client.

```
#include <ros/ros.h>
#include <tutorial_controller/TutorialAction.h>
#include <actionlib/client/simple_action_client.h>
#include "std_msgs/Float64.h"
```

The *tutorial_controller/TutorialAction.h* is the action message library. It'll access the messages created in the .action file.

The *actionlib/server/simple_action_client.h* is the action library used from implementing simple action client. If necessary, you can include other libraries.

Continuing the code, the client class must be set. The action client constructor defines the topic to publish the messages. So, you need specific the same topic name of your server, in this example, *pid_control* was used.

```
class ControllerClient
{
  public:
    ControllerClient(std::string name):

      //Set up the client. It's publishing to topic "
          pid_control", and is set to auto-spin
      ac("pid_control", true),

      //Stores the name
      action_name(name)
      {
    //Get connection to a server
    ROS_INFO("%s Waiting For Server...", action_name.c_str());

    //Wait for the connection to be valid
    ac.waitForServer();

    ROS_INFO("%s Got a Server...", action_name.c_str());

    goalsub = n.subscribe("/cmd_pos", 100, &ControllerClient::
        GoalCallback, this);
  }
```

The *ac.waitForSever()* causes the client waits for the server to start before continuing. Once the server has started, the client informs that established communication with it. Then, define a subscriber (*goalsub*) to provide the setpoint of the control.

The *doneCb* function is called every time that goal completes. It provides state of action server and result message. It'll only be called if the server has not preempted, because the controller must run continuously.

```
28 void doneCb(const actionlib::SimpleClientGoalState& state,
       const tutorial_controller::TutorialResultConstPtr& result){
29   ROS_INFO("Finished in state [%s]", state.toString().c_str());
30   ROS_INFO("Result: %i", result->ok);
31 };
```

The *activeCb* is called every time the goal message is active, in other words, it's called each new goal received by client.

```
33 void activeCb(){
34   ROS_INFO("Goal just went active...");
35 };
```

The *feedbackCb* is called every time the server sends the feedback message to the action client.

```
38 void feedbackCb(const tutorial_controller::
       TutorialFeedbackConstPtr& feedback){
39   ROS_INFO("Got Feedback of Progress to Goal: position: %f",
         feedback->position);
40 };
```

GoalCallback is a function that transmits the goal of the topic */cmd_goal* to the action server.

```
42 void GoalCallback(const std_msgs::Float64& msg){
43   goal.position = msg.data;
44
45   ac.sendGoal(goal, boost::bind(&ControllerClient::doneCb, this
         , _1, _2),
46     boost::bind(&ControllerClient::activeCb, this),
47     boost::bind(&ControllerClient::feedbackCb, this, _1));
48 };
```

The private variables of action client: *n* is a NodeHandle, *ac* is an action client object, *action_name* is a string to set the client name, *goal* is the message that is used to publish goal to server (set in .action file) and *goalsub* is a subscriber to get the goal and pass to the action server.

```
50 private:
51   actionlib::SimpleActionClient<tutorial_controller::
         TutorialAction> ac;
52   std::string action_name;
53   tutorial_controller::TutorialGoal goal;
54   ros::Subscriber goalsub;
```

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

Begin action client:

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

  // create the action client
  // true causes the client to spin its own thread
  ControllerClient client(ros::this_node::getName());

  ros::spin();

  //exit
  return 0;
}
```

4^{th} **step: Create the action server** Therefore, the action client is finished. Now, create action server named *ControllerServer.cpp* in your src folder. Initially, it's necessary include the libraries of ROS, action message and action server. Procedure similar will be made at the client.

```
#include <ros/ros.h>
#include <tutorial_controller/TutorialAction.h>
#include <actionlib/server/simple_action_server.h>
#include "std_msgs/Float64.h"
#include "geometry_msgs/Vector3.h"
#include "sensor_msgs/JointState.h"
#include <math.h>
```

So, we need to set the server class. The action server constructor starts the server. Also, it defines subscriber (feedback loop's control), publisher (PID output), PID limits and initiates the control variables.

```
class ControllerServer{
public:
  ControllerServer(std::string name):
    as(n, "pid_control", boost::bind(&ControllerServer::
        executeCB, this, _1), false),
    action_name(name)
    {
      as.registerPreemptCallback(boost::bind(&ControllerServer
          ::preemptCB, this));

      //Start the server
      as.start();

      //Subscriber current positon of servo
      sensorsub = n2.subscribe("/sensor/encoder/servo", 1, &
          ControllerServer::SensorCallBack, this);

      //Publisher setpoint, current position and error of
          control
```

```
24      error_controlpub = n2.advertise<geometry_msgs::Vector3>("
            /control/error", 1);
25
26      //Publisher PID output in servo
27      controlpub = n2.advertise<std_msgs::Float64>("/motor/
            servo", 1);
28
29      //Max e Min Output PID Controller
30      float max = M_PI;
31      float min = -M_PI;
32
33      //Initializing PID Controller
34      Initialize(min,max);
35    }
```

In the action constructor, an action server is created. A sensor subscriber (*sensor-sub*) and a controller output publisher (*controlpub*) are created to the control loop.

The *preemptCB* informs that the current goal has been canceled by sending a new goal or action client canceled the request.

```
37 void preemptCB(){
38   ROS_INFO("%s got preempted!", action_name.c_str());
39   result.ok = 0;
40   as.setPreempted(result, "I got Preempted!");
41 }
```

A pointer to the goal message is passed in *executeCB* function. This function defines the rate of the controller. You can set the frequency in the *rate* argument of your control. Inside the *while*, the function of the controller should be called, as shown in line 60. It's passed to the controller setpoint (*goal→position*) and the feedback sensor (*position_encoder*).

```
43 void executeCB(const tutorial_controller::TutorialGoalConstPtr&
       goal){
44   prevTime = ros::Time::now();
45
46   //If the server has been killed, don't process
47   if(!as.isActive()||as.isPreemptRequested()) return;
48
49   //Run the processing at 100Hz
50   ros::Rate rate(100);
51
52   //Setup some local variables
53   bool success = true;
54
55   //Loop control
56   while(1){
57     std_msgs::Float64 msg_pos;
58
59     //PID Controller
60     msg_pos.data = PIDController(goal->position,
61         position_encoder);
62
```

```
63      //Publishing PID output in servo
64      controlpub.publish(msg_pos);
65
66      //Auxiliary Message
67      geometry_msgs::Vector3 msg_error;
68
69      msg_error.x = goal->position;
70      msg_error.y = position_encoder;
71      msg_error.z = goal->position - position_encoder;
72
73      //Publishing setpoint, feedback and error control
74      error_controlpub.publish(msg_error);
75
76      feedback.position = position_encoder;
77
78      //Publish feedback to action client
79      as.publishFeedback(feedback);
80
81      //Check for ROS kill
82      if(!ros::ok()){
83        success = false;
84        ROS_INFO("%s Shutting Down", action_name.c_str());
85        break;
86      }
87
88      //If the server has been killed/preempted, stop processing
89      if(!as.isActive()||as.isPreemptRequested()) return;
90
91      //Sleep for rate time
92      rate.sleep();
93    }
94
95    //Publish the result if the goal wasn't preempted
96    if(success){
97      result.ok = 1;
98      as.setSucceeded(result);
99    }
100   else{
101     result.ok = 0;
102     as.setAborted(result,"I Failed!");
103   }
104 }
```

Initialize is a function that sets the initial parameters of a controller. It defines the PID controller output limits and gains.

```
105 void Initialize( float min, float max){
106   setOutputLimits(min, max);
107   lastError = 0;
108   errSum = 0;
109
110   kp = 1.5;
111   ki = 0.1;
112   kd = 0;
```

```
}
```

The *setOutputLimits* function sets the control limits.

```
void setOutputLimits(float min, float max){
  if (min > max) return;
  minLimit = min;
  maxLimit = max;
}
```

The *Controller* function implements the PID equation (Eq. 1). It can be used to design your control algorithm.

```
float PIDController(float setpoint, float PV) {
  ros::Time now = ros::Time::now();
  ros::Duration change = now - prevTime;

  float error = setpoint - PV;

  errSum += error*change.toSec();
  errSum = std::min(errSum, maxLimit);
  errSum = std::max(errSum, minLimit);

  float dErr = (error - lastError)/change.toSec();

  //Do the full calculation
  float output = (kp*error) + (ki*errSum) + (kd*dErr);

  //Clamp output to bounds
  output = std::min(output, maxLimit);
  output = std::max(output, minLimit);

  //Required values for next round
  lastError = error;
  prevTime = now;

  return output;
}
```

Sensor callback is a subscriber that provides sensor information, e.g., position or wheel velocity of a robot. In this case, it receives the position of the servo motor encoder.

```
void SensorCallBack(const sensor_msgs::JointState& msg){
  position_encoder = msg.position[0];
}
```

The protected variables of action server:

```
protected:
  ros::NodeHandle n;
  ros::NodeHandle n2;

  //Subscriber
```

```
157   ros::Subscriber sensorsub;
158
159   //Publishers
160   ros::Publisher controlpub;
161   ros::Publisher error_controlpub;
162
163   //Actionlib variables
164   actionlib::SimpleActionServer<tutorial_controller::
          TutorialAction> as;
165   tutorial_controller::TutorialFeedback feedback;
166   tutorial_controller::TutorialResult result;
167   std::string action_name;
168
169   //Control variables
170   float position_encoder;
171   float errSum;
172   float lastError;
173   float minLimit, maxLimit;
174   ros::Time prevTime;
175   float kp;
176   float ki;
177   float kd;
178 };
```

Finally, the main function creates the action server and spins the node. The action will be running and waiting to receive goals.

```
180 int main(int argc, char** argv){
181   ros::init(argc, argv, "pid_server");
182
183   //Just a check to make sure the usage was correct
184   if(argc != 1){
185     ROS_INFO("Usage: pid_server");
186     return 1;
187   }
188
189   //Spawn the server
190   ControllerServer server(ros::this_node::getName());
191
192   ros::spin();
193
194   return 0;
195 }
```

5^{th} **step: Compile the created package** To compile your controller, it'll be need to add a few things to *CMakeLists.txt*. Firstly, you need to specify the necessary libraries to compile the package. If you require any other library that wasn't mentioned when creating your package, you can add it to *find_package()* and *catkin_package()*.

```
1 find_package(catkin REQUIRED COMPONENTS
2   actionlib
3   actionlib_msgs
4   message_generation
```

```
  roscpp
  rospy
  std_msgs
)

find_package(
  CATKIN_DEPENDS actionlib actionlib_msgs message_generation
      roscpp rospy std_msgs
)
```

Then, specify the action file to generate the messages.

```
add_action_files(
   DIRECTORY action
   FILES Tutorial.action
)
```

And specify the libraries that need .action.

```
generate_messages(
  DEPENDENCIES
  actionlib_msgs std_msgs
)
```

Include directories that your package needs.

```
include_directories(${catkin_INCLUDE_DIRS})
```

The *add_executable()* creates the executable of your server and client. The *target_link_libraries()* includes libraries that can be used by action server and client at build and/or execution. The macro *add_dependencies()* creates a dependency between the messages generated by the server and client with your executables.

```
add_executable(TutorialServer src/ControllerServer.cpp)
target_link_libraries(TutorialServer ${catkin_LIBRARIES})
add_dependencies(TutorialServer ${
    tutorial_controller_EXPORTED_TARGETS} ${
    catkin_EXPORTED_TARGETS})

add_executable(TutorialClient src/ControllerClient.cpp)
target_link_libraries(TutorialClient ${catkin_LIBRARIES})
add_dependencies(TutorialClient ${
    tutorial_controller_EXPORTED_TARGETS} ${
    catkin_EXPORTED_TARGETS})
```

Additionally, the *package.xml* file must include the following dependencies:

```
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<build_depend>message_generation</build_depend>
<run_depend>actionlib</run_depend>
<run_depend>actionlib_msgs</run_depend>
<run_depend>message_generation</run_depend>
```

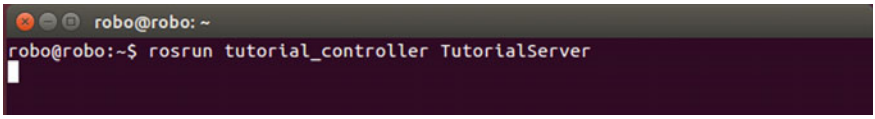

Fig. 5 PID controller server

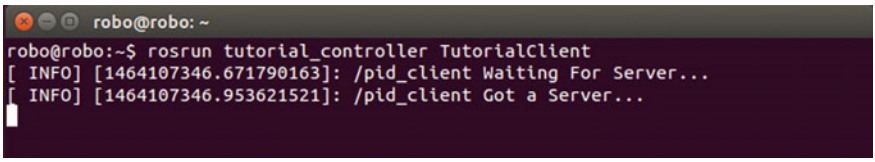

Fig. 6 PID controller client

Now, just compile your workspace:

```
$ cd /home/user/catkin_ws/
$ catkin_make
```

And refresh your ROS environment:

```
$ source /home/user/catkin_ws/devel/setup.bash
```

So your PID controller is ready to be used.

6^{th} **step: Run the controller** Once compiled, your package is ready for use. To run your package, open terminal and start ROS:

```
$ roscore
```

After the ROS starts, you must start the server in new terminal, as shown in Fig. 5.

So, in a new terminal, start the client, as demonstrated in Fig. 6.

PID Controller client waits for the server and notifies you when the connection is established between them.

An alternative to rosrun is *roslaunch*. To use *roslaunch* command, you need to create a folder named *launch* in your package.

```
$ cd /home/user/catkin_ws/src/tutorial_controller
$ mkdir launch
```

After creating the folder, create a launch file (*tutorial.launch*) in the directory launch of your package. In the launch file put the following commands:

```
<launch>
  <node pkg="tutorial_controller" type="TutorialServer" name="TutorialServer"
        output="screen"/>
  <node pkg="tutorial_controller" type="TutorialClient" name="TutorialClient"
        output="screen"/>
</launch>
```

To roslaunch works, it's necessary add roslaunch package in *find_package()* of *CMakeLists.txt*:

```
find_package(
  catkin REQUIRED
  COMPONENTS actionlib actionlib_msgs roslaunch
)
```

And add below line of the *find_package()* in *CMakeLists.txt*:

```
roslaunch_add_file_check(launch)
```

Thus, *CMakeLists.txt* would look like this:

```
cmake_minimum_required(VERSION 2.8.3)
project(tutorial_controller)

find_package(catkin REQUIRED COMPONENTS
  actionlib
  actionlib_msgs
  message_generation
  roscpp
  rospy
  std_msgs
  roslaunch
)

roslaunch_add_file_check(launch)

add_action_files(
  DIRECTORY action
  FILES Tutorial.action
)

generate_messages(
  DEPENDENCIES
  actionlib_msgs std_msgs
)

catkin_package(
  CATKIN_DEPENDS actionlib actionlib_msgs message_generation
      roscpp rospy std_msgs
)

include_directories(${catkin_INCLUDE_DIRS})

add_executable(TutorialServer src/ControllerServer.cpp)
target_link_libraries(TutorialServer ${catkin_LIBRARIES})
add_dependencies(TutorialServer ${
    tutorial_controller_EXPORTED_TARGETS} ${
    catkin_EXPORTED_TARGETS})

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

```
38 add_dependencies(TutorialClient ${
       tutorial_controller_EXPORTED_TARGETS} ${
       catkin_EXPORTED_TARGETS})
```

Save the CMakeLists file. So, you can compile the workspace:

```
1 $ cd /home/user/catkin_ws/
2 $ catkin_make
```

Don't forget to add the workspace to your ROS environment:

```
1 $ source catkin_ws/devel/setup.bash
```

Now, to run your package, you just need this command:

```
1 $ roslaunch tutorial_controller tutorial.launch
```

Please note that *roslaunch* starts ROS automatically, as shown in Fig. 7. Therefore, the *roscore* command isn't required before running the controller.

With *rqt_graph* command, you can see all the topics published and interation between the nodes in ROS.

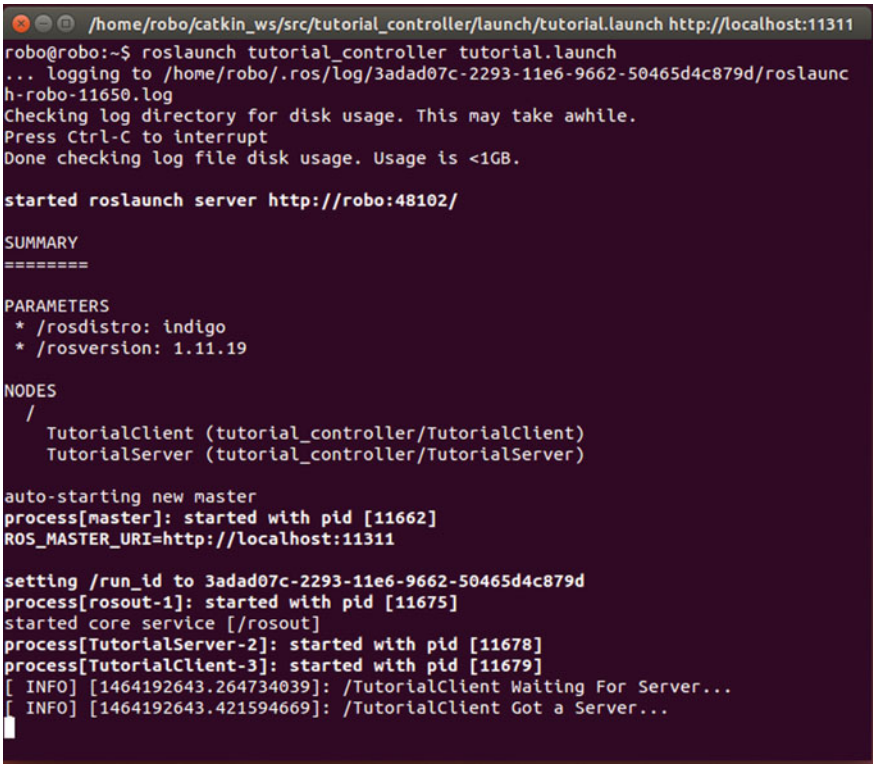

Fig. 7 Roslaunch running PID controller

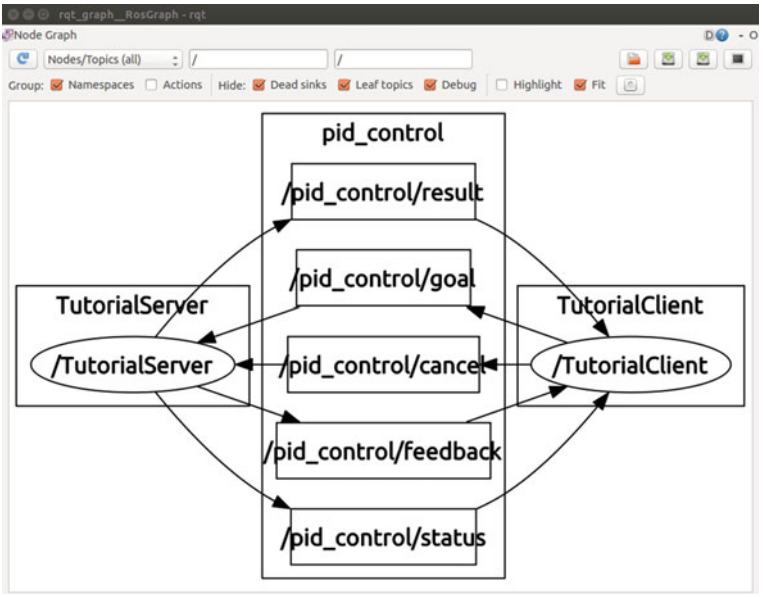

Fig. 8 Interaction between nodes of PID controller

```
$ rqt_graph
```

The rqt_graph package provides a GUI plugin for visualizing the ROS computation graph [7]. You can visualize the topics used for communication between the client and server. Exchanging messages between the client and server are shown in Fig. 8.

The rqt_plot package provides a GUI plugin to plot 2D graphics of the ROS topics [8]. Open new terminal and enter the following command:

```
rqt_plot /control/error/x /control/error/y
```

The variable x is the controller setpoint and y is feedbacked signal (sensor information). In Fig. 9 is shown the rqt_plot plugin.

In case you need, the variable z is the controller error and it can be added in the rqt_plot. Just add the topic */control/error/z* in rqt_plot via command:

```
rqt_plot /control/error/x /control/error/y /control/error/z
```

Or add the error topic directly in the rqt_plot GUI. Just enter the desired topic, as in the Fig. 10, and click on the **+** symbol so that it add more information to plot.

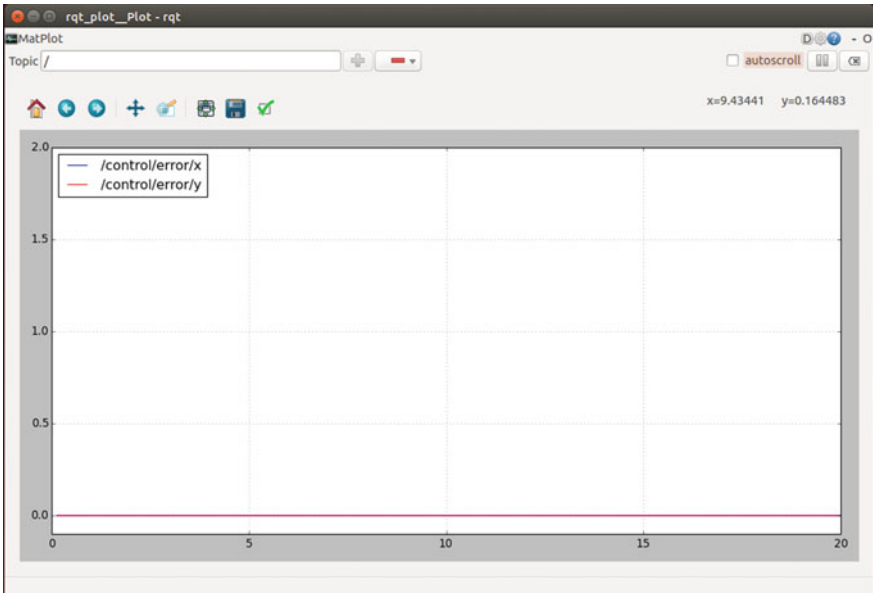

Fig. 9 rqt_plot

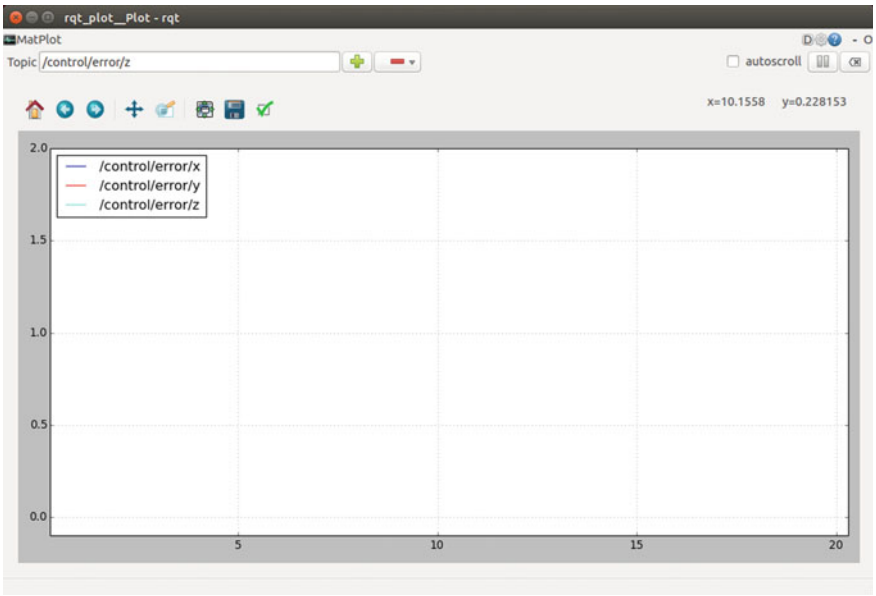

Fig. 10 Add the error topic in rqt_plot

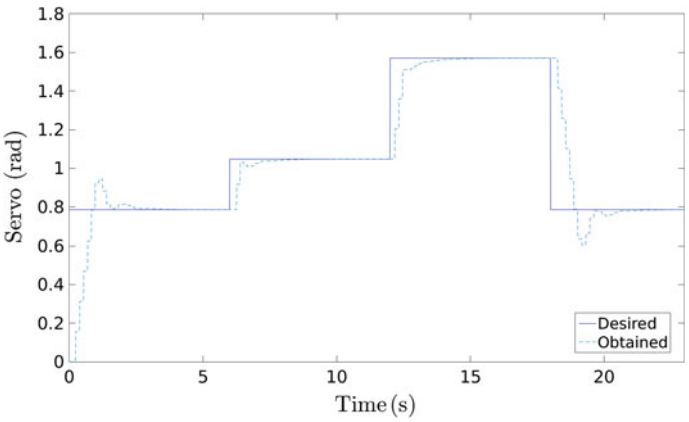

Fig. 11 PID control servo's position

4.2 Experimental Result of PID Controller

For the controller validation, 4 different setpoints were sent to the controller.

The Fig. 11 presents the results of servo position PID control using ActionLib, the control shows a good response.

5 Creating a Fuzzy Controller Using ActionLib

In this section, the implementation of a Fuzzy control using ActionLib will be shown. The fuzzylite library was used to design the fuzzy logic control. The fuzzylite is a free and open-source fuzzy library programmed in C++ for multiple platforms (Windows, Linux, Mac, iOS, Android) [9].

QtFuzzyLite 4 is a graphic user interface for fuzzylite library, you can implement your fuzzy controller using this GUI. Its goal is to accelerate the implementation process of fuzzy logic controllers by providing a graphical user interface very useful and functional allowing you to easily create and directly interact with your controllers [9]. This GUI is available at:

http://www.fuzzylite.com/download/fuzzylite4-linux/.

In the Fig. 12 can be seen the graphic user interface, QtFuzzyLite 4.

In QtFuzzyLite, you can then export the C ++ code to your controller, as shown in Fig. 13.

The Fuzzy controller uses the same application example used in the PID control. In Fig. 14 the fuzzy control diagram of position servo motor can be seen, where β is setpoint, β_s is encoder servo information and u is fuzzy output.

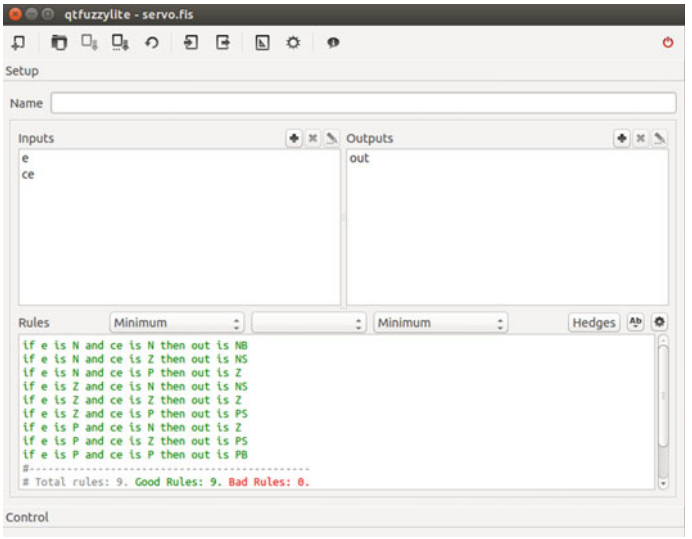

Fig. 12 QtFuzzyLite 4 GUI

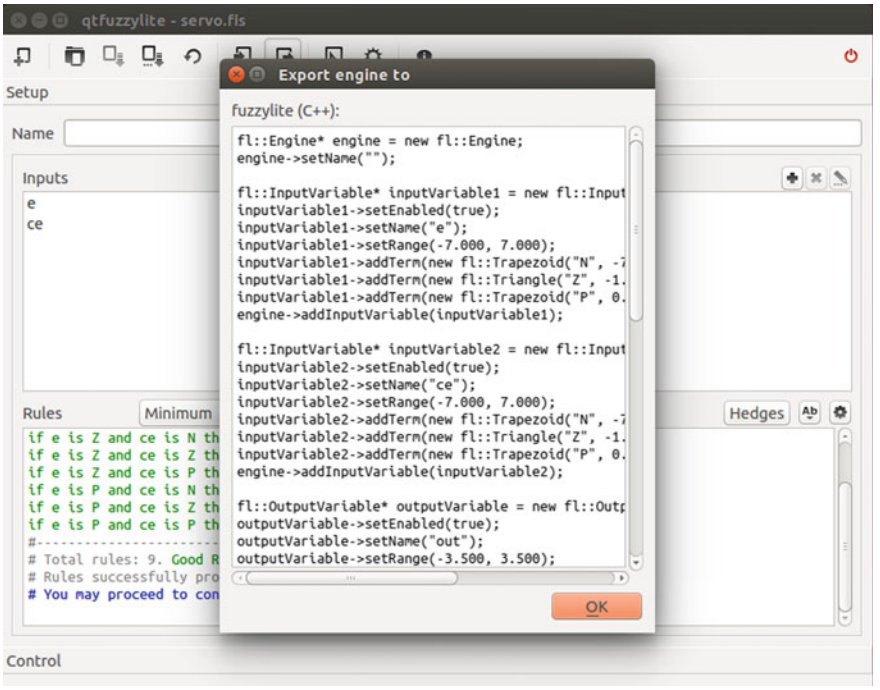

Fig. 13 Export fuzzy to C++ in QtFuzzyLite 4

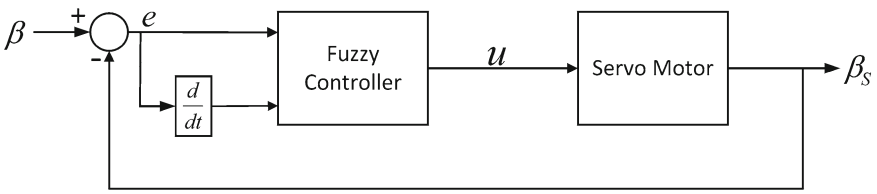

Fig. 14 Fuzzy controller for a servo motor

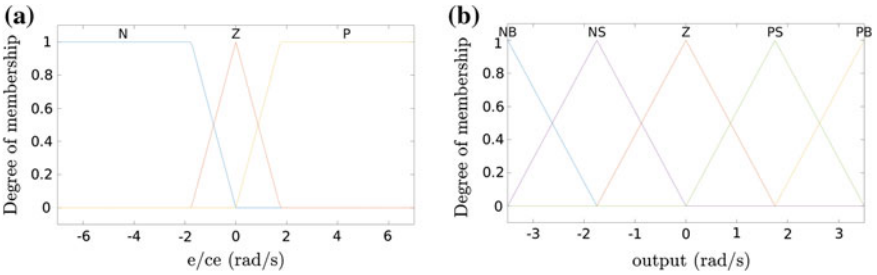

Fig. 15 Membership functions for the servo fuzzy controller of **a** error and change of error and **b** angle increment

The Servo Fuzzy Controller designed for the linear and orientation motion control is presented in Fig. 15. The inputs are 'e' (angle error) and 'ce' (angle change of error), and the output u is angle increment ($u[k] = u[k] + u[k-1]$).

The rules for fuzzy controller are shown in Table 1.

The Fuzzy controller is available on GitHub and can be installed on your workspace:

```
$ source /opt/ros/indigo/setup.bash
$ cd /home/user/catkin_ws/src
$ git clone https://github.com/air-lasca/tutorial2_controller
```

Table 1 Rule table

e / **ce**	N	Z	P
N	NB	NS	Z
Z	NS	Z	PS
P	Z	PS	PB

5.1 Steps to Create the Controller

The creation of Fuzzy controller follows the same steps of the PID. But it has some peculiarities that will be presented below.
1st **step: Creating the ActionLib package** Create the package:

```
$ cd /home/user/catkin_ws/src/
$ catkin_create_pkg tutorial2_controller actionlib
    message_generation roscpp rospy std_msgs actionlib_msgs
```

2nd **step: Creating the action messages** The action file will be the same used by PID controller, but the action's name will be different: *FuzzyControl.action*.
3rd **step: Create action client** The structure of the client will be the same PID client, it will be changed the client's name (*FuzzyClient.cpp*), action (*FuzzyControl.action*), topic (*fuzzy_control*) and package (*tutorial2_controller*).
4th step: **Create the action server** The server will be named *FuzzyServer.cpp*, but you will need to include the fuzzylite library.

```
#include <ros/ros.h>
#include <tutorial2_controller/FuzzyControlAction.h>
#include <actionlib/server/simple_action_server.h>

#include "std_msgs/Float64.h"
#include "geometry_msgs/Vector3.h"
#include "sensor_msgs/JointState.h"
#include <math.h>

//Fuzzylite library
#include "fl/Headers.h"
using namespace fl;
```

The changes mentioned in creating the Fuzzy client should also be made. And the control algorithm will also change, just copy it in the *FuzzyServer.cpp* available on GitHub.
5th **step: Compile the created package** Before you compile the fuzzy controller package, you must copy the *libfuzzylite* library to your */usr/lib/*. Then, add the fuzzylite's source files (*fl* folder) in the *include* directory of your package. The *libfuzzylite.so* file and *fl* folder can be downloaded from the link:
https://github.com/air-lasca/tutorial2_controller.

The *CMakeLists.txt* and *package.xml* follow the instructions specified in the PID controller. Only, in the *CMakeLists.txt*, you'll need to add the *include* folder and add the *libfuzzylite* library, because the server needs to be built.

```
include_directories(${catkin_INCLUDE_DIRS} include)

target_link_libraries(FuzzyServer ${catkin_LIBRARIES}
    libfuzzylite.so)
```

Then, you can compile the package.

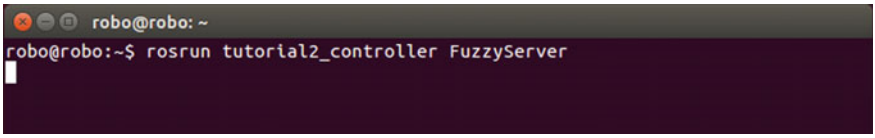

Fig. 16 Fuzzy controller server

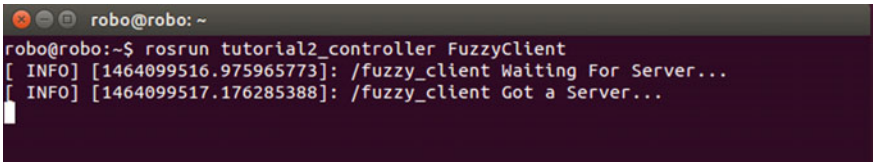

Fig. 17 Fuzzy controller client

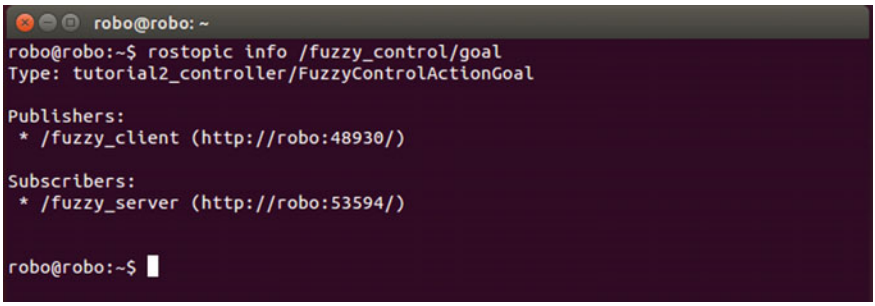

Fig. 18 Getting fuzzy goal information

```
$ cd /home/user/catkin_ws/
$ catkin_make
$ source /home/user/catkin_ws/devel/setup.bash
```

6^{th} **step: Run controller** To run the package, open a terminal and start the ROS.

```
$ roscore
```

Start the server in new terminal. In the Fig. 16, Fuzzy Controller server waits the client.

And start the client in new terminal (Fig. 17).

In Fig. 18 the goal information can be seen: type of message, publisher and subscriber.

The Fig. 19 shows the exchange of messages between the server and client.

You can also use the *roslaunch* to start the controller.

```
$ roslaunch tutorial2_controller fuzzycontrol.launch
```

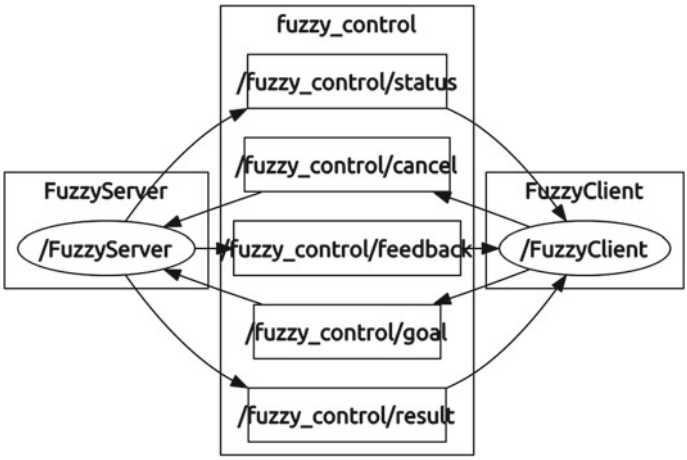

Fig. 19 Interaction between nodes of fuzzy controller

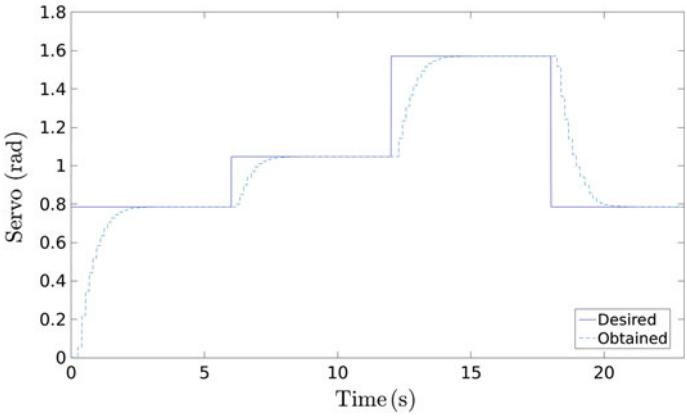

Fig. 20 Fuzzy control servo's position

5.2 Experimental Results of Fuzzy Controller

The results of servo position Fuzzy control are demonstrated in Fig. 20. The fuzzy controller didn't present overshoot in its response curve, even though it had a considerable response time due to the delay of the encoder.

6 Scheduled Fuzzy Controllers for Omnidirectional Motion of an Autonomous Inspection Robot

The scheduled fuzzy controller of AIR-2 is based on the linear velocities $\dot{x}_G$ and $\dot{y}_G$ and angular velocity $\dot{\theta}_G$, as presented in Eq. 3. According to the inputs, switcher (MUX) will choose which controller should be activated. if inputs are the linear velocities, linear motion controller will be activated. Already, when input is only angular velocity, the orientation controller will be enabled. And when inputs are linear and angular velocities, the free motion controller will be activated. The scheduled controllers can be seen in Fig. 21. The experimental results were simulated using V-REP. The control was implement with ActionLib and fuzzylite library.

$$\xi_G = \begin{bmatrix} \dot{x}_G \\ \dot{y}_G \\ \dot{\theta}_G \end{bmatrix} \tag{3}$$

The feedback loop of each controller is related to each control variable. The linear velocities $\dot{x}_R$ and $\dot{y}_R$ of AIR-2 give feedback to linear motion and the angular velocity $\dot{\theta}_R$ of AIR-2 provides feedback to orientation motion. While, linear velocities of AIR-2 and angular velocity $\dot{\beta}_R$ of servo motors give feedback to free motion, due to side slip constraint, AIR-2 can't reorient while moving.

Each motion controller (linear, orientation and free motion) is composed of 8 Fuzzy controllers, in which 4 controllers perform velocity control of brushless motors and other 4 controllers are responsible by angle control of servo motors.

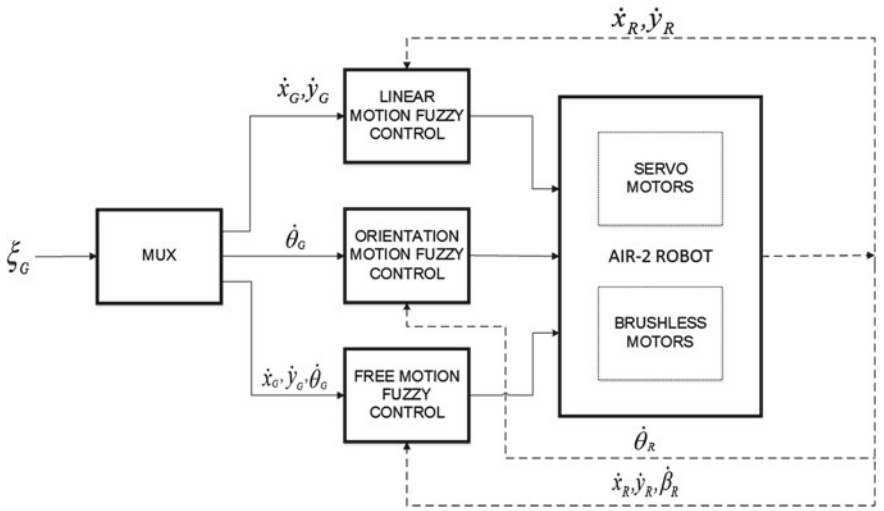

Fig. 21 Scheduled fuzzy controllers of AIR-2

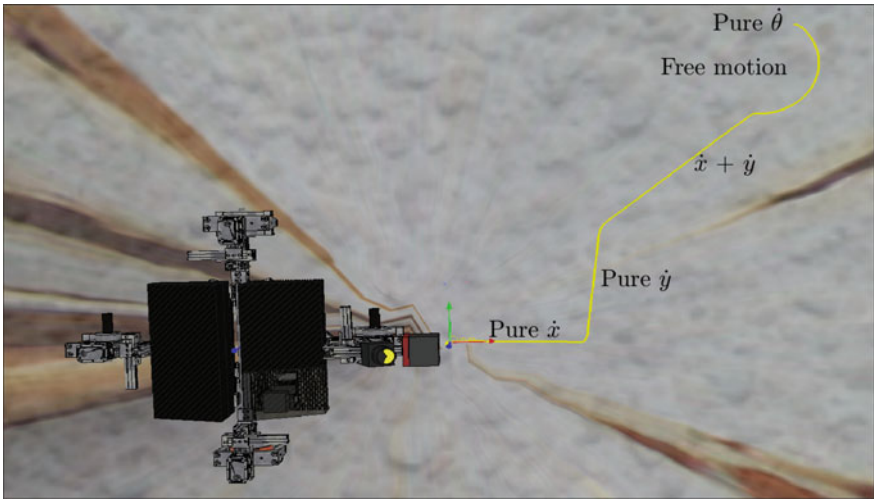

Fig. 22 AIR-2 path in the LPG sphere

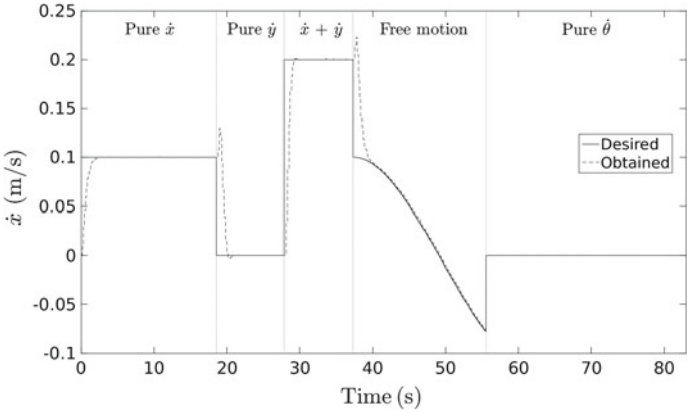

Fig. 23 Desired and obtained $\dot{x}$

A path with five different setpoints is generated for experimental results, which can be seen in Fig. 22.

In first, second and third setpoint, the AIR-2 has been set with linear motion, the setpoint was, respectively, $\dot{x}$, $\dot{y}$ and $\dot{x} + \dot{y}$. The fourth was a free motion with $\dot{x}$ and $\dot{\theta}$. And the fifth setpoint was orientation motion, that means only $\dot{\theta}$.

The low response time of brushless and servo motors produce overshoots that can be seen Figs. 23 and 24. It's caused by sampling frequency of encoders in V-REP.

The Fig. 25 shows the response controller to the $\dot{\theta}$. It has a small oscillation, due data provided by the IMU, even filtered, these data present a great noise. Even so, the control of angular velocity features a good response. In free motion, the high delay of

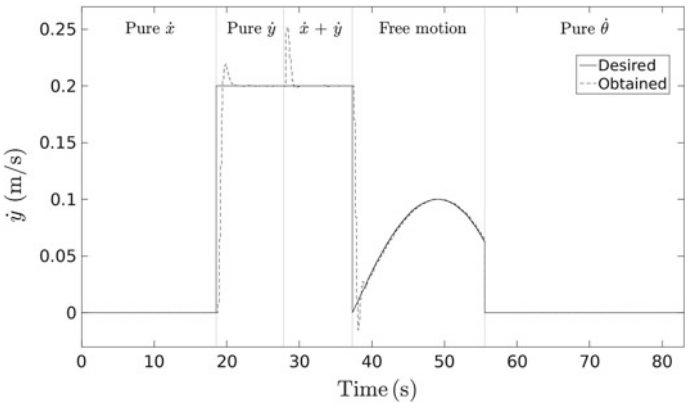

Fig. 24 Desired and obtained $\dot{y}$

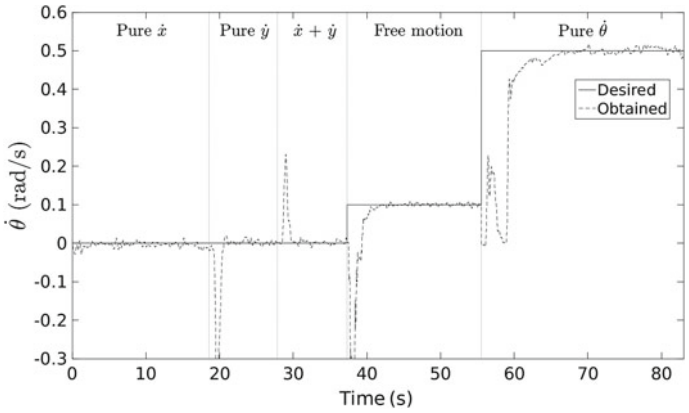

Fig. 25 Desired and obtained $\dot{\theta}$

controller is caused by reorientation wheels. It's necessary to stop brushless motors and servo motors orientate. The servo motors of front and rear are positioned at 90 degrees and left and right are positioned at zero degrees. So, the brushless motors are actuated to control the angular velocity of the AIR-2.

The overshoots and the delay times presented in the speed control don't influence the inspection, since inspection robot operating velocities are low.

The experimental results can be seen in a YouTube video available at the link below:

https://youtu.be/46EKARdyP0w.

7 Conclusion

The ActionLib has proved that the package can be used to implement controllers, exhibited good results as shown in the examples. Easy design of the package allows you to make any adjustments to your control, it allows even implement new control algorithms. The main disadvantage of ActionLib is non-real time, but its preemptive features allow almost periodic execution of the controller.

References

1. Wiki, R. 2016. Actionlib. http://wiki.ros.org/actionlib/.
2. Veiga, R., A.S. de Oliveira, L.V.R. Arruda, and F.N. Junior. 2015. Localization and navigation of a climbing robot inside a LPG spherical tank based on dual-lidar scanning of weld beads. In *Springer Book on Robot Operating System (ROS): The Complete Reference*. New York: Springer.
3. Ren, L., W. Wang, and Z. Du. 2012. A new fuzzy intelligent obstacle avoidance control strategy for wheeled mobile robot. In *2012 IEEE International Conference on Mechatronics and Automation*, 1732–1737.
4. Pratama, D., E.H. Binugroho, and F. Ardilla. 2015. Movement control of two wheels balancing robot using cascaded PID controller. In *International Electronics Symposium (IES)*, 94–99.
5. de Oliveira, A., L. de Arruda, F. Neves, R. Espinoza, and J. Nadas. 2012. Adhesion force control and active gravitational compensation for autonomous inspection in lpg storage spheres. In *Robotics Symposium and Latin American Robotics Symposium (SBR-LARS), 2012 Brazilian*, 232–238.
6. Robotics, C. 2016. Coppelia robotics v-rep: Create. compose. simulate. any robot. http://www.coppeliarobotics.com/.
7. Wiki, R. 2016. rqt_graph. http://wiki.ros.org/rqt_graph.
8. Wiki, R. 2016. rqt_plot. http://wiki.ros.org/rqt_plot.
9. Rada-Vilela, J. 2014. Fuzzylite: a fuzzy logic control library. http://www.fuzzylite.com.