Sensors and Motor Control Lab
Individual Lab Report

Changsheng Shen (Bobby)
Team D (CuBi)

Teammates: Laavanye Bahl, Paulo Camasmie, Jorge Anton Garcia, Nithin Meganathan

February 14, 2019

Individual Progress:

My responsibility in this sensors and motor control lab is to implement:

1.The graphical user interface (GUI)

2.The gateway program for serial communication between the GUI running on the computer and the Arduino microcontroller board that interfaces with the sensor and motors.

Graphical User Interface:

The software framework that I have used to implement the GUI are ROS Qt (rqt), and the dynamic reconfigure package.

ROS Qt is a powerful GUI framework that provides a variety of built-in plugins that is ready to be integrated with ROS-based programs, including plots, tf tree, dynamic reconfigure, etc. With the plots plugin, user can select an active ROS topic, on which data are published from a running program. Then the real-time data will be plotted relative to the timestamp generated from system's clock. It is also straight-forward and convenient to change specific settings regarding the plots, such as line width, color, display scale for x-axis and y-axis, etc.. I have utilized this plugin and have written a program to publish the sensors' data and motor feedback values, received from Arduino through serial port, onto each individual ROS topics. Then in the rqt interface, one can simply select the desired topic name and will be able to see real-time plotting of the data stream.

Below shows the real-time plot of the values of infrared sensor and ultrasonic sensor. Note that the sensors are not calibrated well enough to map to the same physical scale measure in real-world, however both sensors output a similar trend.

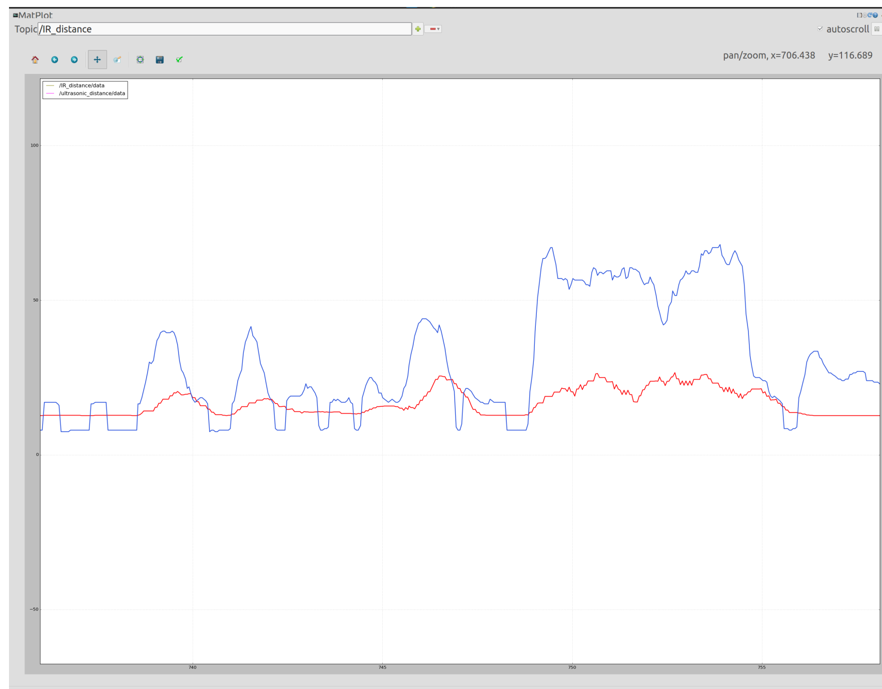

Figure 1. Real-time Plot of Sensor Data

Besides plotting, the GUI also supports sending commands to the Arduino to control the motors' position and velocity. I have written a configuration file and a python function to configure and interface with the dynamic reconfigure plugin package of ROS Qt. Below shows the graphical interface.

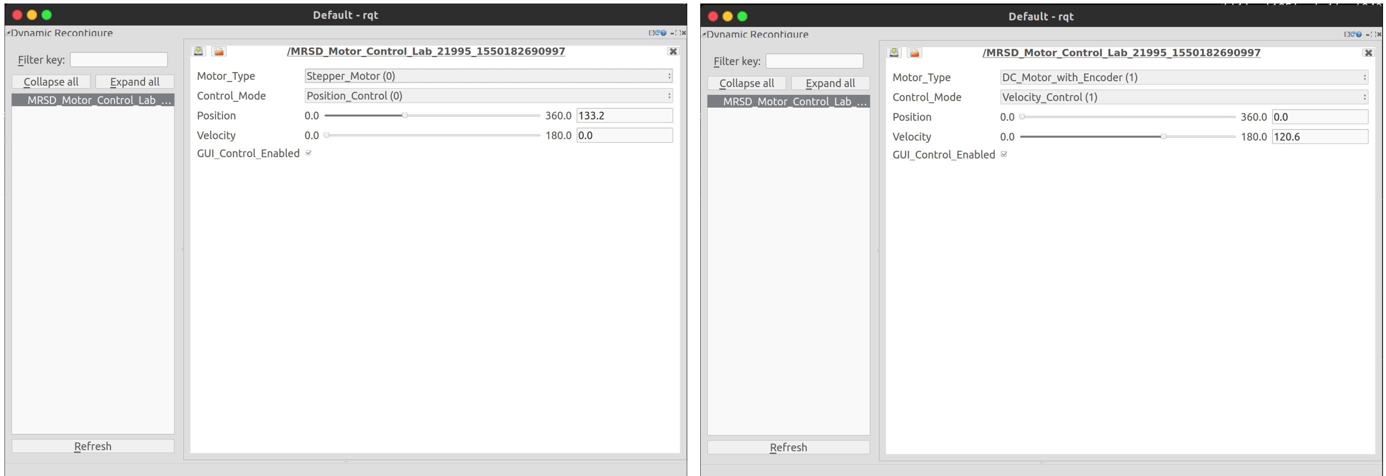

Figure 2. GUI for Sending Commands to Arduino

As shown above, users can select a motor type (stepper motor, DC motor with encoder, or servo motor), a control mode (position or velocity) from the pull-down menu, and then input a value of desired position or velocity. If the "GUI_Control_Enabled" box is checked, the command will be sent to the Arduino through serial communication gateway automatically. The program I have written will also perform a sanity check of the input. It will not send the command if a input command combination is invalid. As an example, the servo motor and stepper motor does not support velocity control mode. Below shows a plot of motor velocity feedback, while the motor's velocity is being controlled by giving multiple different values through the slider bar of the GUI. It can be visualized that the DC motor with encoder is able to respond to the command quickly and relatively accurately.

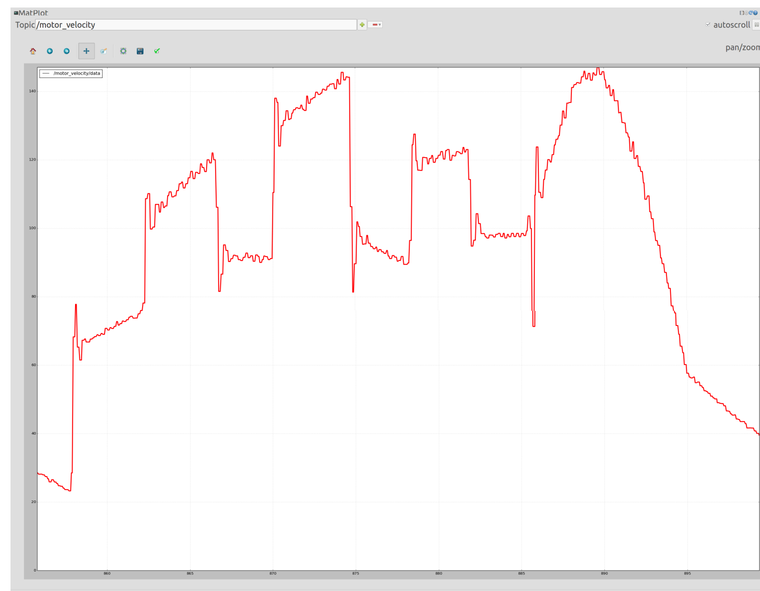

Figure 3. Plot of DC Motor Velocity Feedback with GUI

Serial Communication Gateway:

I have also implemented a serial communication gateway to establish the mutual communication between PC (GUI) and Arduino (sensors and motors).

The program is based on python and ROS serial package. It keeps receiving the data packet from Arduino, decodes it based on the packet protocol we established, and publishes all data onto different ROS topics. On the other hand, it subscribes to the parameter server of ROS. Whenever the parameters for motor commands are changed by user through GUI, it will fetch the updated parameters, pack them into a single comma-separated string, and send it through the serial port to Arduino to control the motors. Codes are attached at the end of this report.

Challenges:

One challenge I have encountered is the configuration and interfacing with the dynamic reconfigure package. In order to achieve a intuitive and easy-to-use user interface, I have to look into the details of the documentation of the plugin, to choose appropriate variable types for pull-down menu selection, slider bar for value adjustment, as well as the check box for enabling or disabling GUI control mode. It also takes some time to figure out the correct way to configure the package and to establish the interface between the python program and the plugin.

Teamwork:

Laavanye Bahl:
For this lab, Laavanye has implemented the program to read the ultrasonic sensor value, apply a moving average low-pass filter, and to control the stepper motor.

Paulo Camasmie:
For this lab, Paulo has implemented the PID controller to control the DC motor with encoder, consisting of both position and velocity feedback control.

Jorge Anton Garcia:
For this lab, Jorge implemented the program to drive servo motor, to read the IR distance sensor and the photomicrosensor, and to perform button debouncing. He also implemented the communication protocol on the arduino-side and did the hardware-software integration.

Nithin Meganathan:
For this lab, Nithin wrote the transfer function for the sensors to map their voltage reading to real-world physical quantities. He also built the circuits for the force sensor and servo motor.

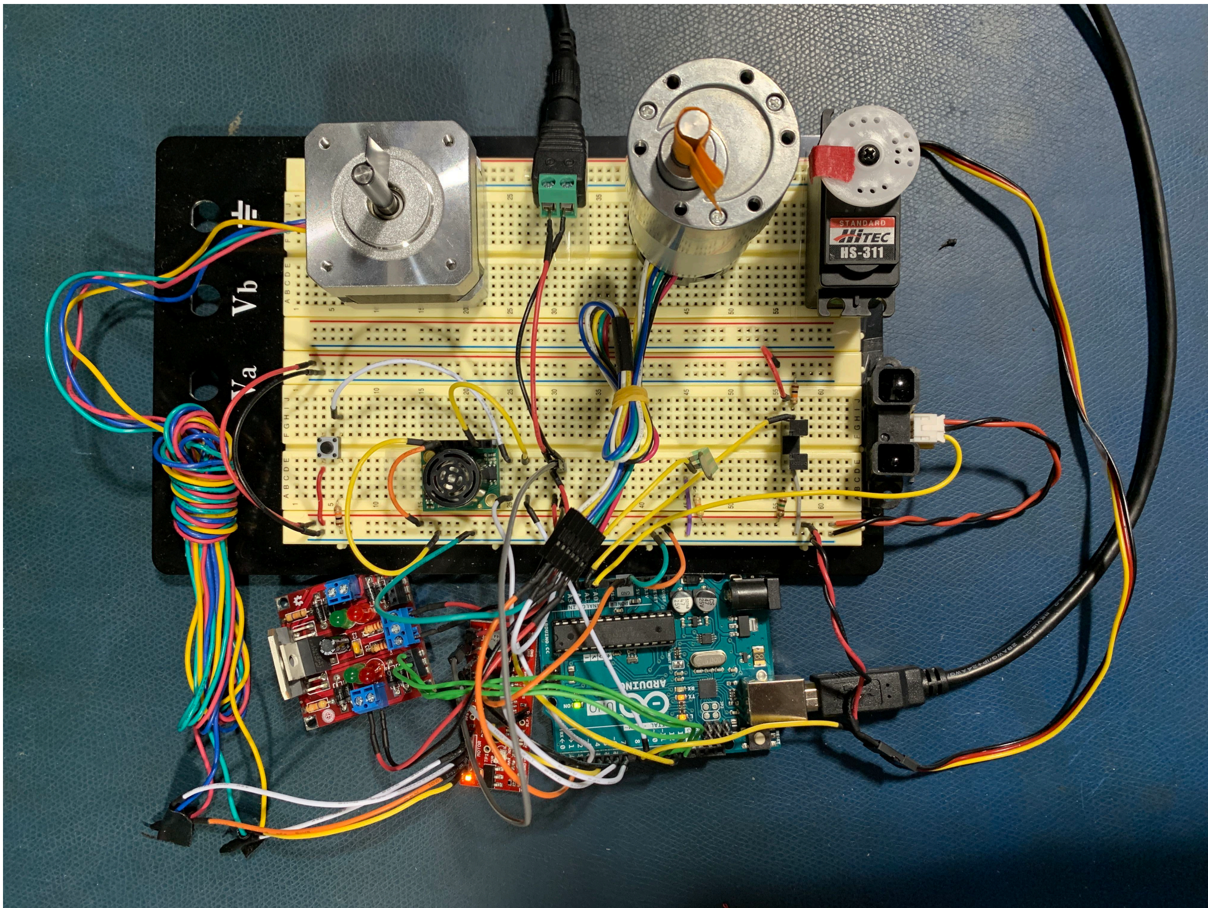

Figure 4. The Final Circuit Integration for all Sensors and Motors

Project Progress:

For the CuBi project, we assembled the first two layers of TurtleBot 3 Waffle Pi as our mobile platform. An Arduino-compatible controller board and a Raspberry Pi 3 was mounted and connected to the motors on the chassis. Software development environments were setup such that we can drive the mobile platform either by using a wireless joystick, or by sending commands through ROS over the wifi network.

For the grasping part, we designed and printed several prototypes of the grippers. We tested them on the toys that we are aiming to pick up, and iterated the design. Next step will be to integrate the gripper with actuators, and install it onto the mobile platform to test. We also started to implement software for object segmentation for grasping, based on RGBD cameras.

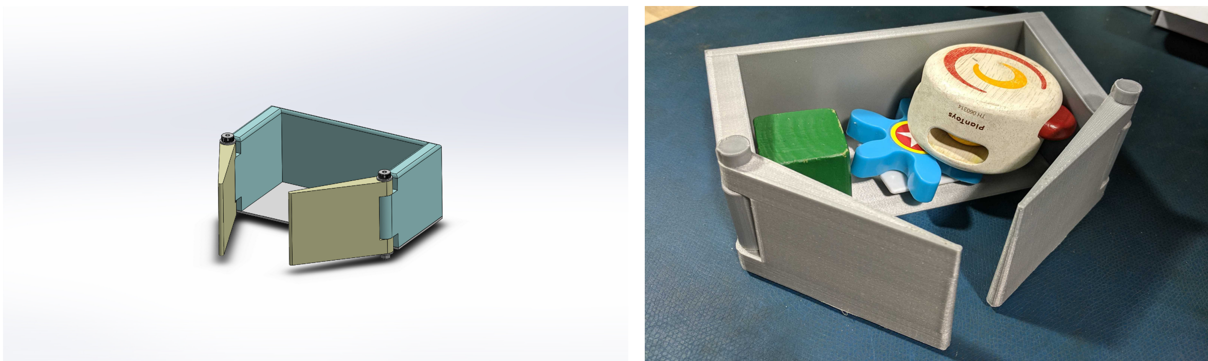

Figure 5. The Design and Prototype of the Gripper

Quiz:

1. ADXL335 Accelerometer

1. **Range:** $\pm 3g$

2. **Dynamic range:** $6g$

3. **Purpose of C_{DC}:** Filtering out the high-frequency component of input voltage source, make it "cleaner". For high-frequency component, this capacitor will act as a wire that shorts the high-frequency component, such that it will not affect the input voltage supply

4. **Transfer function:** $V_{out} = 1.5V + \frac{0.3}{g} \cdot a$

5. **Largest expected nonlinearity error in g:** $\pm 0.3\% * 6g = \pm 0.018g$

6. **Noise when excited at 25 Hz:** $150\mu g * \sqrt{25} = 750\mu g$

7. **Noise when excited at 0 Hz:** We can determine this by measuring the output voltage at zero input, then subtract it with the mean value (zero g bias, 1.5V for this sensor) to get the noise.

2. Signal conditioning:

Filtering:

1. **Problem with moving average:** not robust to outliers, introduces a lag in time domain of the output signal relative to the raw input signal (the larger the window size, the larger the delay is).

2. **Problem with median filter:** not robust to high-frequency noise, such as gaussian random noise.

Opamps:

1. **Input range -1.5V to 1.0V:**

Choose V_1 as the reference voltage, V_2 as the input voltage. Denote $x = \frac{R_f}{R_i}$, then we have the equation:

$$V_{out} = V_2(1 + x) - V_1 \cdot x$$

Plug in the values and solve the two equations, we can get: $V_1 = 3V, \frac{R_f}{R_i} = 1$

Similarly, if we choose V_2 as the reference voltage, V_1 as the input voltage. We have:

$$V_{out} = -V_1 \cdot x + V_2(1 + x)$$

Solving this gives: $V_2 = -3V, \frac{R_f}{R_i} = -2$. However, this is an invalid solution since resistance cannot be negative.

2. Input range -2.5V to 2.5V:

The calibration cannot be done with this circuit. If we choose V_1 as the reference voltage, V_2 as the input voltage, similar to above, solving the equation gives $\frac{R_f}{R_i} = -1$, which is invalid.

If we choose V_2 as the reference voltage, V_1 as the input voltage, solving the equations gives $\frac{R_f}{R_i} = 0$. In this case, $V_{out} = V_{ref} = V_2$ which is a constant voltage. Therefore, the calibration cannot be done with this circuit.

3. Control

The PID controller consists of a proportional gain, an integral term, and a derivative term. The final control output u at timestamp t is calculated as the sum of the three terms:

$$u_t = K_p \cdot error_t + K_i \cdot error_t \cdot \Delta t + K_d \cdot (error_t - error_{t-1})/\Delta t$$

where $error_t$ is defined as the difference between target position and current position at timestamp t, and time interval Δt is the time difference (in seconds) between two adjacent timestamps. A boundary limit is also introduced to prevent the integral term from "winding-up" too much.

If the system is sluggish, I will increase the proportional term (P term) because it directly decreases the system's rise time.

After applying this, if the system still has significant steady-state error, I will increase the integral term (I term), because it integrates error over time and accumulate control output based on that, thus it helps eliminate the steady-state error.

After this, if the system still has overshoot, I will increase the derivative term (D term) because the derivative term anticipates the error, which tends to add damping to the system and therefore reduces overshoot.

Code (serial communication gateway):

```
#!/usr/bin/env python

import time
import serial
import struct
from Queue import *
import math
from math import sin, cos, pi

import rospy
import tf
import roslib

# import numpy as np

from std_msgs.msg import String,Float32, Int8
from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3

# from threading import Thread

from dynamic_reconfigure.server import Server
from mrsd_motor_control_lab.cfg import motorConfig

obstacle_pub = rospy.Publisher('obstacle', Int8, queue_size=1)
force_pub = rospy.Publisher('force', Float32, queue_size=10)
dist_IR_pub = rospy.Publisher('IR_distance', Float32, queue_size=10)
dist_ultrasonic_pub = rospy.Publisher('ultrasonic_distance', Float32, queue_size=10)

motor_vel_pub = rospy.Publisher('motor_velocity', Float32, queue_size=10)
motor_pos_pub = rospy.Publisher('motor_position', Float32, queue_size=10)

rospy.init_node('MRSD_Motor_Control_Lab', anonymous=True)
rate = rospy.Rate(100) # 100hz

# current_time = rospy.Time.now()
# last_time = rospy.Time.now()

serial_port = serial.Serial()

# Motor type:
# 0 - Stepper Motor
# 1 - DC Motor with Encoder
# 2 - Servo Motor
motor_type = 0

# Control mode:
# 0 - Position control
# 1 - Velocity control
control_mode = 0

# Control value:
# angle (0 to 360 deg) for position control
# velocity (RPM) for velocity control (for DC Motor with encoder only)
control_value = 0.0
```

```
data_length = 6

dist_IR_prev = 0.0
obstacle_buffer_size = 5
obstacle_buffer = [1.0, 1.0, 1.0, 1.0, 1.0]

def updateParams(motor_type, control_mode, control_value):

    motor_command = str(motor_type) + "," + str(control_mode) + "," +
str(int(control_value)) + ","
    print(motor_command)

    if(serial_port.isOpen()):
        serial_port.write(motor_command)
    else:
        print("Error: Serial port " + str(serial_port.name) + " is not open")

def paramCallback(config, level):

    gui_enable = config["GUI_Control_Enabled"]

    motor_type = config["Motor_Type"]

    if motor_type == 0:
        motor = "Stepper Motor"
    elif motor_type == 1:
        motor = "DC Motor with Encoder"
    elif motor_type == 2:
        motor = "Servo Motor"

    control_mode = config["Control_Mode"]

    position = config["Position"]
    velocity = config["Velocity"]

    if gui_enable:
        if control_mode == 0:
            updateParams(motor_type, control_mode, position)
        elif control_mode == 1:

            if motor_type == 1:
                updateParams(motor_type, control_mode, velocity)
            else:
                print("Invalid: " + motor + " does not support velocity control
mode!")
    else:
        print("GUI Control not enabled...")

    return config

def myShutdownProc():

    serial_port.close()
    print("Serial port (" + serial_port.name + ") still on is " +
str(serial_port.isOpen()))
    print "Serial talker shutdown!"
    rospy.signal_shutdown("END Serial Talker")
```

```
def serial_receiving(serial_port):


    serial_port.flush()

    # Get current ROS time
    # last_time = current_time
    # current_time = rospy.Time.now()

    serial_incoming_raw = serial_port.readline()
    incoming_data = [i for i in serial_incoming_raw.split(",")]

    # for item in incoming_data:
    #     print(item)

    print(incoming_data)

    if len(incoming_data) == data_length:


        obstacle = incoming_data[0] # 0: is obstacle, 1: no obstacle
        force = incoming_data[1] # N
        dist_IR = incoming_data[2] # cm
        dist_ultrasonic = incoming_data[3] # mm

         #dist_ultrasonic = float(dist_ultrasonic)/10.0 # convert to cm

        # motor_id = incoming_data[4] # 0: stepper, 1: dc motor with encoder, 2: servo
        # data_type = incoming_data[5] # 0: position, 1: velocity
        # value = incoming_data[6] # degrees for position, RPM for velocity
        dc_motor_pos = incoming_data[4]
        dc_motor_vel = incoming_data[5]

        # print(obstacle)
        # print(force)
        # print(dist_IR)
        # print(dist_ultrasonic)
        msg_obstacle = Int8()
        msg = Float32()


        global obstacle_buffer, obstacle_buffer_size

        avg = 0.0
        for i in range(obstacle_buffer_size - 1):
            obstacle_buffer[i] = obstacle_buffer[i+1]
            avg += obstacle_buffer[i]
        obstacle_buffer[obstacle_buffer_size-1] = float(obstacle)
        avg += float(obstacle)
        avg /= obstacle_buffer_size
        if avg < 0.7:
            msg_obstacle.data = 0
        else:
            msg_obstacle.data = 1
        obstacle_pub.publish(msg_obstacle)
```

```
        msg.data = float(force)
        force_pub.publish(msg)

        msg.data = float(dist_IR) * 0.5 + dist_IR_prev * 0.5
        dist_IR_pub.publish(msg)
        global dist_IR_prev
        dist_IR_prev = float(dist_IR)

        msg.data = float(dist_ultrasonic) / 10.0
        dist_ultrasonic_pub.publish(msg)

        msg.data = float(dc_motor_pos)
        motor_pos_pub.publish(msg)

        msg.data = float(dc_motor_vel)
        motor_vel_pub.publish(msg)
        # msg.data = value
        # if data_type == 0: # position
        #     motor_pos_pub.publish(msg)
        # elif data_type == 1: # velocity
        #     motor_vel_pub.publish(msg)



            #rospy.logerr("data interpertation error")
    else:
        print("Data length error!")
        print("Expected data length = " + str(data_length))
        print("Received data length = " + str(len(incoming_data)))


def talker():
    rospy.on_shutdown(myShutdownProc)

    serial_port.baudrate = 9600
    serial_port.timeout = None
    serial_port.port = '/dev/ttyACM0' # need to be changed based on hardware setting.
    print(serial_port)
    serial_port.open()

    print(serial_port.isOpen())
    print(serial_port.name)
    print(serial.VERSION)

    while not rospy.is_shutdown():

        serial_receiving(serial_port)

        rate.sleep()

if __name__ == '__main__':
    srv = Server(motorConfig, paramCallback)
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Code (configuration file for dynamic reconfigure):

```
#!/usr/bin/env python

PACKAGE = "mrsd_motor_control_lab"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()

motor_type_enum = gen.enum([ gen.const("Stepper_Motor", int_t, 0, “stepper motor"),
                       gen.const("DC_Motor_with_Encoder",     int_t, 1, “dc motor"),
                       gen.const("Servo_Motor",      int_t, 2, “servo motor")],
                       "An enum to select motor type")

gen.add("Motor_Type", int_t, 0, "A motor type parameter which is edited via an enum",
0, 0, 3, edit_method=motor_type_enum)

control_mode_enum = gen.enum([ gen.const("Position_Control", int_t, 0, "position"),
                       gen.const("Velocity_Control",     int_t, 1, "velocity")],
                       "An enum to set control mode")
gen.add("Control_Mode", int_t, 0, "A control type parameter which is edited via an
enum", 0, 0, 2, edit_method=control_mode_enum)

gen.add("Position", double_t, 0, "desired position for the motor", 0.0,  0, 360)
gen.add("Velocity", double_t, 0, "desired velocity for the motor", 0.0,  0, 180)

gen.add("GUI_Control_Enabled", bool_t, 0, "Enable GUI Control Mode", False)

exit(gen.generate(PACKAGE, "motor_command", "motor"))
```