Instituto Tecnológico y de Estudios Superiores de Monterrey

TE3001B.101

Fundamentación de robótica (Gpo 101)

Semestre: Febrero - Junio 2022

Reto semanal 2 Manchester Robotics

Alumnos:

José Jezarel Sánchez Mijares	A01735226
Antonio Silva Martínez	A01173663
Frida Lizett Zavala Pérez	A01275226

Fecha de entrega: 24 de Febrero del 2023

Reto semanal 2. Manchester Robotics

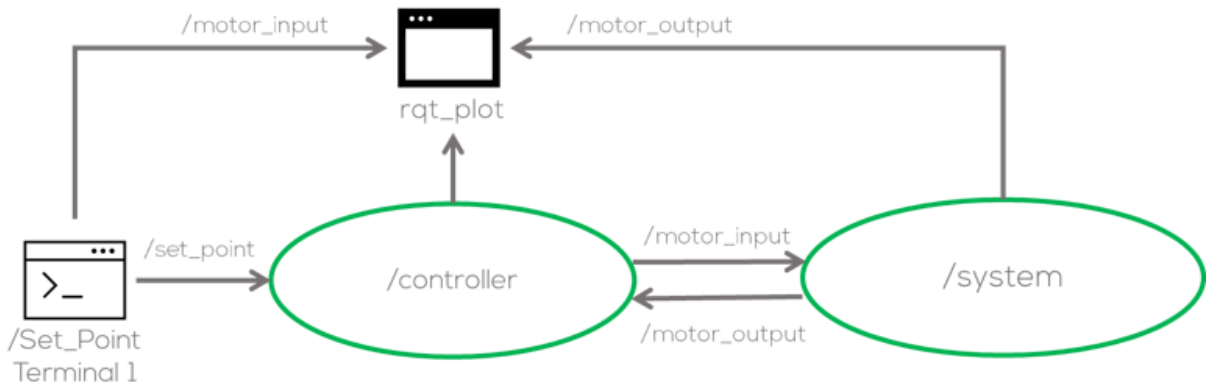
Resumen

Este mini desafío está enfocado para que el estudiante repase los conceptos introducidos en las sesiones previas. La actividad consiste en crear un controlador para un sistema de primer orden simple en ROS. El sistema representa el comportamiento dinámico de un motor de corriente continua (DC).
El nodo "/system" y una estructura de programa simple (no obligatoria) son proporcionados por MCR2.
El controlador puede ser un controlador "P", "PI" o "PID" (otros controladores pueden ser aceptados previa consulta con el profesor).

Objetivos

Desarrollar en el alumno las capacidades de manejo del framework ROS, y con los conocimientos adquiridos en la actividad previa, incrementar el grado de interacción y la implementación de funciones disponibles en ROS.

Este reto permitirá al alumno comprender el funcionamiento básico de un controlador para interactuar con la respuesta generada por un motor (a nivel simulación) a través de ROS.



Introducción

En ROS, los namespaces son una forma de organizar los nodos y tópicos de manera jerárquica. Los namespaces permiten evitar conflictos de nombres y agrupar tópicos y servicios relacionados. Por ejemplo, se podría tener un namespace "robot 1" que contenga nodos, tópicos y servicios relacionados con un robot específico, y otro namespace "robot 2" para otro robot.

Los parámetros son valores que se pueden pasar a los nodos y ajustar en tiempo de ejecución. Los parámetros pueden ser utilizados para configurar un nodo o cambiar su comportamiento, sin necesidad de recompilar o reiniciar el nodo. Los parámetros también pueden ser utilizados para compartir información entre nodos.

Los mensajes personalizados (Custom messages) son definiciones de mensajes que se pueden utilizar para enviar y recibir información entre nodos en ROS. Los mensajes personalizados permiten definir un formato específico para los datos que se envían a través de los tópicos, servicios y acciones en ROS. Por ejemplo, se podría definir un mensaje personalizado "Pose2D" que contenga la posición y orientación de un robot en un plano 2D. Los mensajes personalizados se definen utilizando el lenguaje de definición de mensajes de ROS (msg), y se pueden utilizar en cualquier nodo que implemente ese mensaje.

Solución del problema

Para la preparación de la realización del reto de esta semana fue necesario utilizar un controlador el cual pudiera adaptarse a un sistema el cual estaba controlado por un controlador, en donde nosotros teníamos que diseñar este mismo para que los valores resultantes fueran acorde a la medición que nosotros quisiéramos hacer.

En este reto se utilizaron diversas herramientas, desde python, hasta programas en matlab para poder modelar el sistema conforme a los parámetros que necesitaremos para modelos en el sistema que nosotros desarrollamos, siendo que en esta ocasión para la solución de este problema se decidió utilizar un controlador PI debido a que esta es una opción popular para el control de motores DC debido a su simplicidad y efectividad en el control de la velocidad y la posición del motor. El controlador PI combina las características del controlador proporcional y el integral, lo que permite una respuesta rápida y precisa en la regulación de la velocidad del motor, al tiempo que reduce el error de estado estable.

El controlador proporcional (P) utiliza la señal de error actual para producir una señal de control proporcional al error, lo que significa que cuanto mayor sea el error, mayor será la señal de control. El controlador integral (I) utiliza la integral del error a lo largo del tiempo para producir una señal de control que disminuye el error de estado estable. La combinación de estos dos controladores en el controlador PI permite una respuesta rápida y precisa del motor, al tiempo que minimiza el error de estado estable.

set_point generator.py

```
#!/usr/bin/env python
import rospy
import numpy as np
from pid_control.msg import set_point

# Configurar las variables, parámetros y mensajes a utilizar (si
es necesario).

#Stop Condition
def stop():
 #Configurar el mensaje de parada (puede ser el mismo que el
mensaje de control).
  print("Stopping")


if __name__=='__main__':
    #Se comprueba que el archivo se está ejecutando directamente y
no se está importando como un módulo.
    #Se inicializa un nodo ROS con el nombre
"Set_Point_Generator".
    #Se define una tasa de ejecución de 100 Hz (0.01 segundos de
intervalo de tiempo) utilizando el objeto "Rate" de ROS.
    rospy.init_node("Set_Point_Generator")
    rate = rospy.Rate(100)
    rospy.on_shutdown(stop)

    #Se define la función "stop" como el callback que se llamará
cuando el nodo se detenga.
    pub = rospy.Publisher("/set_point", set_point, queue_size = 1)
```

```
    #Se configura el objeto "Publisher" de ROS para enviar
mensajes de "set_point" en el tópico "/set_point".
    print("The Set Point Genertor is Running")

    #Se imprime un mensaje en la consola indicando que el nodo se
está ejecutando y se inicia un bucle que se
    #ejecutará mientras el nodo no se haya detenido.
    while not rospy.is_shutdown():

        #Se define la variable "sp" como el valor del parámetro
"sp" definido en el nodo ROS.
        #Si no se define ningún valor para el parámetro "sp", se
utilizará el valor por defecto de 8.0.
        sp = rospy.get_param("sp", 8.0)

        #Se crea un nuevo objeto de mensaje "set_point" y se
asigna el valor de "sp" a la propiedad "outsp" del mensaje.
        outputsp = set_point()
        outputsp.outsp = sp

        rospy.loginfo(outputsp)
        #pub.publish(outputsp)

        #Se utiliza la función "loginfo" de ROS para imprimir el
mensaje de set_point en la consola. A continuación, se publica el
mensaje utilizando el objeto "Publisher" definido anteriormente.
        rate.sleep()
```

Controlador.py

```
#!/usr/bin/env python
#from typing import Self

#Este codigo fue complementado y tomando de esqueleto al codigo
# realizado en las clases de Control y Actuadores con el arduino
import rospy
import numpy as np
from pid_control.msg import motor_output
from pid_control.msg import motor_input
from pid_control.msg import set_point
```

```
#Setup parameters, variables and callback functions here (if
required)

#Definicion de las variables en motor_input.msg
input = motor_input()
input.input = 0.0
input.time = 0.0

#Parametros de control_params.yaml para diferentes valores de
entrada
VMotor = rospy.get_param("sp", 0.0)
VMotor1 = rospy.get_param("sp1", 5.0)
VMotor2 = rospy.get_param("sp2", 1.0)
VMotor3 = rospy.get_param("sp3", 8.0)
VMotor4 = rospy.get_param("sp4", 3.0)
VMotor5 = rospy.get_param("sp5", 7.0)
VMotor6 = rospy.get_param("sp6", 7.0)

#Diferentes valores de kp, ki, kd para ver el desempeno del
controlador
kp = rospy.get_param("kp", 2.0)

ki = rospy.get_param("ki", 6.0)

#Tiempo de muestreo
Ts = rospy.get_param("TS", 0.02)

#Definicion de las variables en motor_output.msg
output = motor_output()
output.output = 0.0
output.time = 0.0
output.status = ""


#Calculo de constantes del controlador PID con diferentes
situaciones(subamortiguado, criticamente,sobreamortiguado)
K1 = kp + Ts*ki

u = [0.0, 0.0]        #U es la Matriz de salida del controlador
e = [0.0, 0.0, 0.0]   #E es la Matriz de error del sistema
controlado
```

```
#Funcion para definir las variables de salida
def callback(msg) :
  global output
  output = msg

def callbacksp(msgsp):
  global set_sp
  set_sp = msgsp.outsp

#Stop Condition
def stop():
 #Setup the stop message (can be the same as the control message)
  print("Stopping")

if __name__=='__main__':
      #Initialise and Setup node
      rospy.init_node("controller")
      rate = rospy.Rate(100)
      rospy.on_shutdown(stop)

      #Setup Publishers and subscribers here
      pub = rospy.Publisher("/motor_input", motor_input, queue_size
= 1)
      rospy.Subscriber("/motor_output", motor_output, callback)

      rospy.Subscriber("/set_point", set_point, callbacksp)

      print("The Controller is Running")
      #Run the node
      while not rospy.is_shutdown():
      #Write your code here

      #Se muestran las posibles diferentes
situaciones(subamortiguado, sobreamortiguado y criticamente
amortiguado)
      #Para ello se hizo variar los valores de kp,ki, kd y el
voltaje de entrada
      if (output.time < 10):
            e[0] = VMotor - output.output
      if(output.time > 10.1):
            e[0] = VMotor1 - output.output
      if(output.time > 15):
```

```
            e[0] = VMotor2 - output.output

            kp
            ki

            Ts

            K1 = kp + Ts*ki

      if(output.time > 25):
            e[0] = VMotor3 - output.output

            kp
            ki
            Ts

            K1 = kp + Ts*ki

      if(output.time > 35):
            e[0] = VMotor4 - output.output

            kp
            ki
            Ts

            K1 = kp + Ts*ki
      #En esta condicion se deja el controlador mal entonado, y en
la grafica se ve como queda oscilando
      if(output.time > 45):
            e[0] = VMotor5 - output.output

            kp
            ki

            Ts

            K1 = kp + Ts*ki

      if(output.time > 55):
            e[0] = VMotor6 - output.output

            kp
```

```
            ki

            Ts

            K1 = kp + Ts*ki


      u[0] = K1*e[0] + K1*e[1] + K1*e[2] + u[1]
      e[2] = e[1]
      e[1] = e[0]
      u[1] = u[0]

      input.input = u[0] * (1.0 / 13.0)

      pub.publish(input)
      print(output.output)

      rate.sleep()
```

system.py

```
#!/usr/bin/env python
import rospy
import numpy as np
from pid_control.msg import motor_output
from pid_control.msg import motor_input
from std_msgs.msg import Float32


#En esta sección se importan los módulos necesarios para el
correcto funcionamiento del nodo.
#En particular, se importan los módulos rospy, numpy y std_msgs,
así como los mensajes
#personalizados motor_input y motor_output.
class SimpleSystem:

  #Se define una clase llamada SimpleSystem, que contendrá todos
los métodos necesarios para la simulación del sistema.
  def __init__(self):

    #Establecer los parámetros del sistema.
    self.sampleT = rospy.get_param("/system_sample_time",0.02)
```

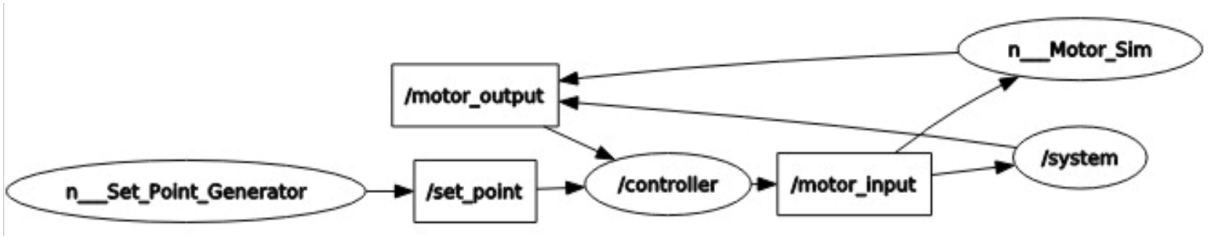
```
    self.speedmx = rospy.get_param("/system_max_speed",13.0)
    self.inputmn = rospy.get_param("/system_min_input",0.2)
    self.p_K = rospy.get_param("/system_param_K",13.2)
    self.p_T = rospy.get_param("/system_param_T",0.05)
    self.init_conditions =
rospy.get_param("/system_initial_cond",0)

    #Configurar las variables a utilizar.
    self.first = True
    self.stime = 0.0
    self.ctime = 0.0
    self.ltime = 0.0
    self.p_output = 0.0

    #Declarar el mensaje de entrada.
    self.Input = motor_input()
    self.Input.input = 0.0
    self.Input.time = 0.0


    #Declarar el mensaje de salida del proceso.
    self.output = motor_output()
    self.output.output= self.init_conditions
    self.output.time = rospy.get_time()
    self.MotorStatus(self.init_conditions)


    # Configura el Subscribers

rospy.Subscriber("/motor_input",motor_input,self.input_callback)

    #Configura el publishers
    self.state_pub = rospy.Publisher("/motor_output",
motor_output, queue_size=1)

  #En el método __init__, se inicializan las variables del sistema
y se declaran los mensajes de entrada y salida.
  #Además, se configuran los suscriptores y publicadores
correspondientes. En particular,
  #el nodo se suscribe al tópico /motor_input, que se utilizará
para recibir la señal de entrada del sistema,
  #y publica en el tópico /motor_output, que se utilizará para
```

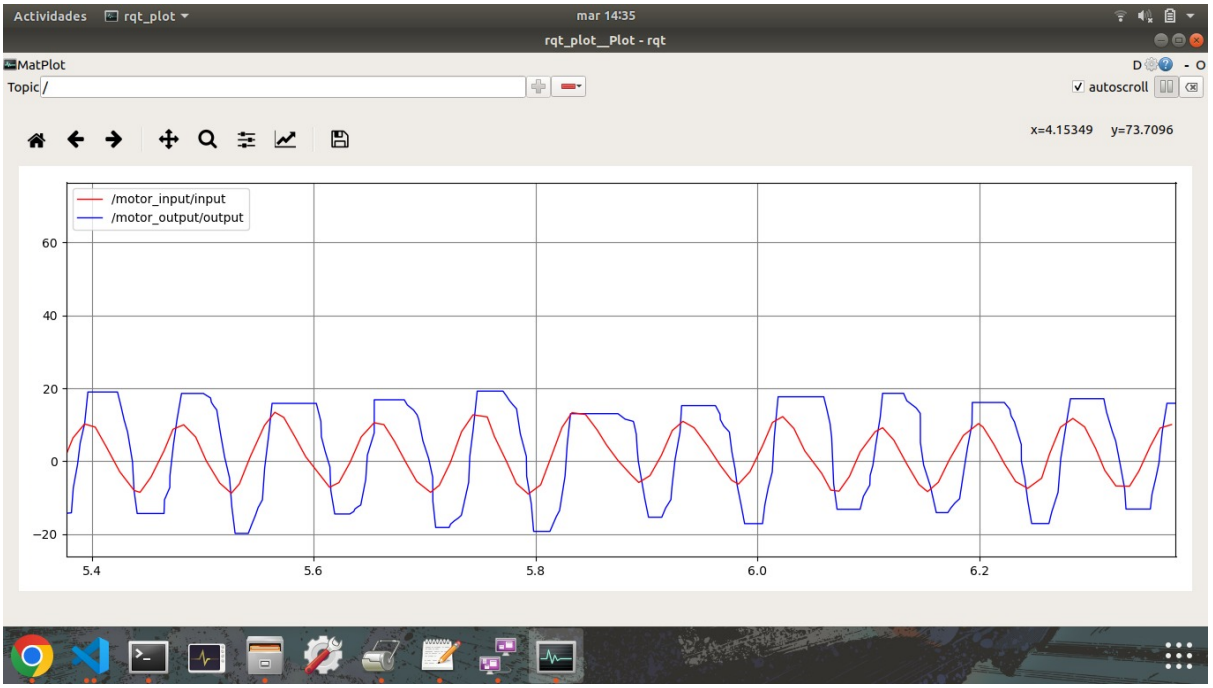
```
enviar la señal de salida del sistema.

  #Define el callback
  def input_callback(self,msg):
    self.Input = msg

  #Se define una función de callback para el mensaje de entrada
del sistema.
  #Esta función simplemente actualiza el valor del mensaje Input
con el valor del mensaje recibido.
  #Define the main RUN function
  def run (self):
    #Configuración de variables.
    if self.first == True:
      self.stime = rospy.get_time()
      self.ltime = rospy.get_time()
      self.ctime = rospy.get_time()
      self.first = False
  #System
    else:
      #Define sampling time
      self.ctime = rospy.get_time()
      dt = self.ctime - self.ltime

      #Simulación del sistema dinámico.
      if dt >= self.sampleT:
        #Dead-Zone
        if(abs(self.Input.input)<=self.inputmn):
          self.p_output+= (-1.0/self.p_T * self.p_output +
self.p_K/self.p_T * 0.0) * dt
        #Saturacion
        elif (((-1.0/self.p_T * self.p_output + self.p_K/self.p_T
* self.Input.input)>0.0 and self.p_output> self.speedmx)or
((-1.0/self.p_T * self.p_output + self.p_K/self.p_T *
self.Input.input)<0.0 and self.p_output< -self.speedmx)):
          self.p_output+= (-1.0/self.p_T * self.p_output +
self.p_K/self.p_T * ((1/self.p_K)*self.p_output)) * dt
        #Sistema dinámico.
        else:
          self.p_output += dt*((-1.0/self.p_T) * self.p_output +
(self.p_K/self.p_T) * self.Input.input)
```

```
        #Mensaje a publicar.
        self.output.output= self.p_output
        self.output.time = rospy.get_time() - self.stime
        self.MotorStatus(self.p_output)
        #Publicar mensaje.
        self.state_pub.publish(self.output)

        self.ltime = rospy.get_time()

      #else:
      #self.state_pub.publish(self.output)

  #Función de estado del motor.
  def MotorStatus(self,speed):
    if (abs(speed)<=abs(self.p_K*self.Input.input*0.8) and
abs(self.Input.input)<=self.inputmn):
      self.output.status = "Motor Not Turning"
    elif (abs(speed)>=self.speedmx):
      self.output.status = "Motor Max Speed"
    else:
      self.output.status = "Motor Turning"



 #Stop Condition
  def stop(self):
  #Configura el mensaje de parada
    print("Stopping")
    self.output.output= 0.0
    self.output.time = rospy.get_time() - self.stime
    self.output.status = "Motor Not Turning"
    self.state_pub.publish(self.output)
    total_time = rospy.get_time()-self.stime
    rospy.loginfo("Total Simulation Time = %f" % total_time)



if __name__=='__main__':

 #Initializa el nodo
 rospy.init_node("Motor_Sim")
 System = SimpleSystem()
```

```
# Configura el nodo
loop_rate = rospy.Rate(rospy.get_param("/system_node_rate",100))
rospy.on_shutdown(System.stop)

print("The Motor is Running")
try:
 #Corre el nodo
 while not rospy.is_shutdown():
  System.run()
  loop_rate.sleep()

except rospy.ROSInterruptException:
 pass
```

Resultados





Los resultados obtenidos fueron los esperados ya que el controlador que se escogió fue el adecuado y por ende la señal resultante fue una señal que se adapta a los cambios que ocurrían con el tiempo,

Conclusiones

En la robótica, los Namespaces, Parameters y Custom messages son elementos esenciales en el desarrollo de aplicaciones robustas y flexibles. Los Namespaces permiten organizar los nodos y tópicos de manera jerárquica, lo que simplifica la gestión de múltiples robots o sistemas complejos. Los “Parameters” permiten ajustar el comportamiento de los nodos y compartir información entre ellos, lo que es especialmente útil en aplicaciones de control en tiempo real. Los Custom messages permiten definir un formato específico para los datos que se envían entre los nodos, lo que simplifica el intercambio de información y facilita la integración de diferentes componentes de la aplicación.

En resumen, los Namespaces, Parameters y Custom messages son herramientas clave en la robótica para desarrollar aplicaciones escalables, modulares y adaptables. Su uso adecuado permite simplificar el desarrollo y mantenimiento de los sistemas robóticos, y facilita la integración de diferentes componentes y tecnologías. Por lo tanto, es importante que los desarrolladores de aplicaciones robóticas comprendan y utilicen estos conceptos de manera efectiva para lograr un éxito óptimo en el desarrollo de aplicaciones robóticas avanzadas.

Referencias

1. ROS Wiki. (2023). recuperado el 27 de febrero del 2023, desde http://wiki.ros.org/es
2. ROS: Home. (2023). recuperado el 27 de febrero del 2023, desde https://www.ros.org/
3. Quigley, M., Gerkey, B., & Smart, W. D. (2015). Programming robots with ROS: A practical introduction to the Robot Operating System. O'Reilly Media, Inc.