Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Puebla

TE3003B.501

Integración de robótica y sistemas inteligentes

Reto Semanal 3 | Manchester Robotics

Frida Lizett Zavala Pérez A01275226

Diego Garcia Rueda A01735217

Alejandro Armenta Arellano A01734879

26 de Abril del 2024

Índice

Resumen

Se implementó la técnica de linealización del modelo cinemático no lineal del robot no holonómico para el análisis y simulación del Puzzelbot en RViz. Este proceso implica utilizar la expansión de Taylor, específicamente los jacobianos, para aproximar el modelo no lineal alrededor de un punto de operación. Al linealizar el modelo cinemático, se simplifica para su análisis y control mediante técnicas diseñadas para sistemas lineales. Esto permite una mejor comprensión del comportamiento del robot y facilita el diseño de controladores más eficientes y precisos para el movimiento y la navegación.

Objetivos

Linealizar el modelo cinemático no lineal del robot no holonómico utilizando la expansión de Taylor (jacobianos) para facilitar el análisis y comprensión del comportamiento del robot, permitiendo el diseño de controladores más efectivos adaptados a sistemas lineales, mejorar la precisión del control y seguimiento de trayectorias, lo anterior con la finalidad de facilitar la simulación y validación del controlador antes de su implementación definitiva, lo que contribuye a optimizar el diseño y desempeño del sistema robótico.

Introducción

Una motivación para la linealización es que el comportamiento dinámico de muchos sistemas no lineales dentro de un rango de variables puede ser aproximado a modelos de sistemas lineales. Siendo ese el caso, se pueden usar técnicas bien desarrolladas de análisis y síntesis de sistemas lineales para analizar un sistema no lineal. La linealización se introdujo con la finalidad de obtener un modelo aproximado debido a que linealizar la ecuación alrededor de

un punto de interés nos proporciona un modelo aproximado que captura el comportamiento esencial del sistema en ese punto.

Linealización de sistemas dinámicos

La linealización es un procedimiento que permite aproximar un modelo no lineal, por otro que sí lo es y que cumple con las propiedades de los sistemas lineales, en particular el principio de superposición. Esto implica simplificar el modelo no lineal del robot alrededor de un punto de operación, generalmente un punto de equilibrio o una configuración particular, para que pueda ser analizado y controlado de manera más efectiva usando técnicas de control lineal.

El proceso de linealización de un sistema dinámico de un robot diferencial consta de los siguientes pasos principales:

- Determinación del punto de operación: Se selecciona un punto de operación alrededor del cual se realizará la linealización. Este punto puede ser un punto de equilibrio o una configuración específica del robot. Se necesitan valores para las velocidades de las ruedas y posiblemente otras variables de estado para definir este punto de operación.
- Modelado del sistema: Se utilizan las ecuaciones cinemáticas y dinámicas del robot para modelar el comportamiento, las cuales no son lineales debido a la relación entre las velocidades de las ruedas y la orientación del robot.
- Linealización alrededor del punto de operación: Se hace la aproximación de las ecuaciones no lineales mediante las series de Taylor (Jacobianos), donde se conservan sólo los términos lineales. Considerando el punto de operación definido, obteniendo como resultado un modelo lineal del sistema que describe el comportamiento del robot cuando está cercano al punto de operación.

- Análisis y control lineal: Al obtener el modelo linealizado se le pueden realizar análisis de sistemas lineales para diseñar controladores y analizar la estabilidad y rendimiento del sistema.

Localización basada en el movimiento (Dead Reckoning)

La localización de un robot móvil basada en el modelo cinemático del sistema es conocido como Dead-Reckoning. Una manera sencilla de estimar la posición de un robot consiste en monitorear las variables de estado del robot (x, y, theta) usando sensores como giroscopios, acelerómetros, tacómetros y codificadores rotativos; este proceso estima la posición del robot basándose en la velocidad y dirección del vehículo, considerando el tiempo transcurrido desde la última posición conocida hasta la actual. Teniendo así información únicamente desde donde se inició la navegación del robot. Uno de los principales inconvenientes es que el error estimado de posición se incrementa con el tiempo ya que se basa en el estimado anterior.

Variables aleatorias discretas y continuas

Las variables aleatorias son aquellas que asumen un valor numérico único para cada uno de los resultados que aparecen en el espacio muestral de un experimento de probabilidad. Dicho de otra manera, la variable aleatoria es una función que asocia un número real con cada elemento del espacio muestral.

Existen dos tipos de variables aleatorias discretas y continuas siendo las primeras las que tienen un número finito de valores o un número de valores contable a diferencia de las variables aleatorias continuas que puede tomar cualquier valor numérico dentro de un intervalo, de modo que entre cualesquiera dos de ellos siempre existe otro posible valor.

Linealización en torno a los puntos de equilibrio

La linealización tiene como propósito usar un sistema lineal para predecir el comportamiento de las soluciones de un sistema no lineal cerca de un punto de equilibrio, en dicho punto las variables de estado del sistema permanecen constantes todo el tiempo. Esto permite ver el comportamiento de un sistema dinámico cerca de condiciones estables y cómo responde a las perturbaciones del entorno. Siendo útil en términos de este proyecto para diseñar estrategias de control que aseguren que el sistema permanezca en su punto de equilibrio o se mueva hacia él de manera precisa y rápida.

Solución del problema

Para el modelo linearizado se decidió modificar la arquitectura utilizada en los anteriores retos del socio formador, específicamente se modificó el nodo *Real_Sim_Robot* que es el encargado de tener el modelo cinemático del robot, teniendo aquí el modelo cinemático es donde se tiene que realizar el proceso de linealización en esta sección, el desarrollo seguido para la implementación de la elaboración se hizo mediante el desarrollo de las ecuaciones diferenciales del modelo cinemático del *Puzzlebot*

$$\dot{x} = V\cos(\theta) \tag{1}$$

$$\dot{y} = V\sin(\theta) \tag{2}$$

$$\dot{\theta} = \omega \tag{3}$$

Las ecuaciones (1) a (3) muestran el modelo cinemático del; robot en un plano 2D, al ser en 2 dimensiones es por eso que la velocidad lineal *V* tiene componentes x,y. Para la obtención de los valores de velocidad lineal y angular utilizamos las ecuaciones (4) y (5), velocidades obtenidas a partir de las velocidades angulares de cada llanta del *Puzzlebot.*

$$V = \frac{(W_R * R + W_L * R)}{2} \tag{4}$$

$$W = \frac{(W_L * R - W_R * R)}{L} \quad (5)$$

Donde *R* es el radio de las ruedas mientras que *L* es la longitud de la separación entre las ruedas. El siguiente procedimiento de desarrollo es la obtención de las ecuaciones de salida del sistema de espacio de estados, debido a que la entrada del sistema son las velocidades lineales y angulares la salida del sistema son las posiciones, las ecuaciones de salida del sistema son las siguientes:

$$x = x(t_0) + \Delta t * v * cos(\Theta) \quad (6)$$

$$y = y(t_0) + \Delta t * v * sen(\Theta) \quad (7)$$

$$\Theta = \Theta(t_0) + \Delta t * \omega \quad (8)$$

Es mediante las ecuaciones de salida del sistema y las variables de estado quien se procede con la elaboración de un jacobiano utilizando las variables de estado que son las posiciones del robot y orientación de sí mismo. Se parte con la fórmula (9) que describe al Jacobiano siendo las ecuaciones de salidas derivadas parcialmente con las variables del sistema. Al ser 3 ecuaciones de salida y 3 variables de estado el desarrollo del Jacobiano queda descrito como en la ecuación (10) mostrando cada una de las derivadas parciales. En la ecuación (11) se sustituyen las funciones de salida y se declara la variable de estado en cada derivada parcial resultando en el jacobiano utilizado para la linealización.

$$\frac{dy(x)}{dt} = \frac{dy(x)}{dx} * \frac{dx}{dt} \quad (9)$$

$$\frac{dy(x)}{dt} = \frac{dy(x)}{dX_1} * \frac{dX_1}{dt} + \frac{dy(x)}{dX_2} * \frac{dX_2}{dt} + \frac{dy(x)}{dX_3} * \frac{dX_3}{dt} \quad (10)$$

$$\frac{dF}{dx}(x) = \begin{bmatrix} x(t_0) + \Delta t * v * cos(\Theta) & y(t_0) + \Delta t * v * sen(\Theta) & \Theta(t_0) + \Delta t * \omega \end{bmatrix} \begin{bmatrix} x \\ y \\ \Theta \end{bmatrix} \quad (11)$$

El resultado del Jacobniano se muestra en (12), es mediante este jacobiano que podemos realizar la linealización en el código.

$$\begin{bmatrix} 1 & 0 & -\Delta t * V * sin(\Theta)) \\ 0 & 1 & \Delta t * V * cos(\Theta)) \\ 0 & 0 & 1 \end{bmatrix} \tag{12}$$

Para la inclusión de la linealización se decidió modificar el nodo *Real_Sim_Robot* en la función *update_Pose (apéndice **Real_Sim_Robot**)*, esta función es la encargada de calcula la velocidad lineal y angular (v,w) a partir de las velocidades angulares de cada llanta, estas líneas de código son análogas de las ecuaciones (4) y (5). Este nodo al publicar un mensaje de tipo *PoseStamped* es que se tiene que declarar los valores que adquirieron los componentes de este mensaje (posición, orientación), la función *update_Pose* se encarga de esto igualmente, reemplazando las líneas de código que establecen valores de posición y orientación del modelo no linealizado por los valores del modelo linealizado usando como base el Jacobiano desarrollado en (12). Los 2 modelos se encuentran mostrados en la imagen (1).

```
def update_Pose(self, dt):
    v = self.wheel_radio * (self.wl + self.wr) / 2.0
    w = self.wheel_radio * (self.wr - self.wl) / self.wheel_base

    #sistema no linealizado
    self.robot_pose.pose.position.x += v * math.cos(self.robot_pose.pose.orientation.z) * dt
    self.robot_pose.pose.position.y += v * math.sin(self.robot_pose.pose.orientation.z) * dt
    self.robot_pose.pose.orientation.z += w * dt

    #sistema linealizado
    #self.robot_pose.pose.position.x = -v * np.sin(self.robot_pose.pose.orientation.z) * self.robot_pose.pose.position.x * dt
    #self.robot_pose.pose.position.y = v * np.cos(self.robot_pose.pose.orientation.z) * self.robot_pose.pose.position.y * dt
    #self.robot_pose.pose.orientation.z = self.robot_pose.pose.orientation.z + w * dt
```

Imagen 1. Sección de código mostrando modelos linealizado y no linealizado

Resultados

Funcionamiento de la simulación en *Rviz* con el modelo linealizado.

https://youtu.be/WhPyl6odC9M

En el video se logra apreciar como el modelo linealizado presenta buenos resultados siguiendo la trayectoria indicada (un cuadrado de longitud de 1 metro por lado) siendo resultado colaborativo por parte del control del robot así como el proceso de linealización, sin

embargo cabe destacar que el modelo linealizado no presenta cambios tan notorios a comparación del modelo no linealizado, como hipótesis por parte del equipo se intuye que es debido al controlador PID del robot. Respecto al reto anterior se logró concluir con el desarrollo del controlador PID mejorándolo hasta llegar a un grado de precisión de 99.8% siendo este un gran factor de que las diferencias entre el modelo linealizado y no linealizado sean mínimas.

Conclusiones

La linealización en torno a puntos de equilibrio simplifica modelos no lineales para su análisis cerca de condiciones estables. Este proceso es esencial para comprender la estabilidad del sistema y diseñar un control efectivo.

Bibliografía o referencias

Variables discretas y continuas – Alianza B@UNAM, CCH & ENP ante la pandemia. (2024, 8 febrero). https://alianza.bunam.unam.mx/cch/variables-discretas-y-continuas/

Franco, O. G. (2023, 30 junio). *Ecuaciones Diferenciales I: Linealización de los puntos de equilibrio de sistemas no lineales*. El Blog de Leo. https://blog.nekomath.com/ecuaciones-diferenciales-i-linealizacion-de-los-puntos-de-equilibrio-de-sistemas-no-lineales/

Khan Academy. (s.f.). Local Linearization. Recuperado de https://es.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/tangent-planes-and-local-linearization/a/local-linearization

CK-12 Foundation. (s. f.). *CK-12 Foundation*. https://flexbooks.ck12.org/cbook/c%C3%A1lculo-2.0/section/4.10/primary/lesson/linearizaci%C3%B3n-de-una-funci%C3%B3n-calc-spn/

Joseph, A. (2014). The History of Measuring Ocean Currents. En *Elsevier eBooks* (pp. 51-92). https://doi.org/10.1016/b978-0-12-415990-7.00002-8

Dead Reckoning (DR). (2022, 22 junio). SKYbrary Aviation Safety. https://skybrary.aero/articles/dead-reckoning-dr

Apéndice Real_Sim_Robot

```
#!/usr/bin/env python3
import rospy
import numpy as np
from geometry_msgs.msg import Twist, Point, Vector3, Quaternion,
Pose
from nav_msgs.msg import Odometry


class ControlNode:
      def _init_(self):
            self.odom_sub = rospy.Subscriber('/odom', Odometry,
self.odom_callback)
            self.cmd_pub = rospy.Publisher('/cmd_vel', Twist,
queue_size=10)
            #Publishers for Debugging Goal Coordinate and Error of
PID
            self.ref_pub = rospy.Publisher('/ref', Twist,
queue_size=10)
            self.error_publisher = rospy.Publisher('/errors',
Twist, queue_size=10)

            #Physical attributes of the Puzzlebot
            self.wheel_base = 0.19
            self.wheel_radio = 0.05

            self.kp_lin = rospy.get_param("linear_kp", 0.5)
            self.kp_ang = rospy.get_param("angular_kp", 0.82)

            self.ki_lin = rospy.get_param("linear_ki", 0.008)
            self.ki_ang = rospy.get_param("angular_ki", 0.0002)

            self.kd_lin = rospy.get_param("linear_kd", 0.05)
            self.kd_ang = rospy.get_param("angular_kd", 0.05)

            #Accumulative Errors
            self.error_acc_ang = 0.0
            self.error_acc_lin = 0.0


            #Intregral Errors
```

```
            self.error_int_ang = 0.0
            self.error_int_lin = 0.0

            #Last Linear and Angular Errors
            self.error_prev_ang = 0.0
            self.error_prev_lin = 0.0

            # Reference (goal) - Dummy Init
            self.local_goal: Point = None

            # Initial state (inertial frame)
            self.odom_pose : Pose = Pose(position = Point(x = 0.0,
y = 0.0, z = 0.0),

orientation=Quaternion(x=0.0, y=0.0, z=0.0, w=1.0))


            #Points of the trajectory
            self.goals = [Point(x = 1.0, y = 0.0, z = 0.0),
                          Point(x = 1.0, y = 1.0, z = 0.0),
                    Point(x = 0.0, y = 1.0, z = 0.0),
                    Point(x = 0.0, y = 0.0, z = 0.0)]


            #Index to iterate through the coordinate list
            self.index = 0
            #Limit of coordinates
            self.n_goals = len(self.goals)
            #Function to create the NextPoint-Trajectory
            self.set_Trajectory(self.goals[self.index])

            self.last_med_ang = 0.0
            self.last_med_lin = 0.0

            #Tolerance for Distance and Angle error
            #Angular tolerance change from degrees to Radians
            self.ang_tolerance : float = 0.009
            self.lin_tolerance : float = 0.009

            self.error_ang = 0.0
            self.error_lin = 0.0
```

```
            #Pose attributes for Localisation
            self.pose_x = rospy.get_param("pose_x", 0.0)
            self.pose_y = rospy.get_param("pose_y", 0.0)
            self.pose_z = 0.0
            self.pose_theta = rospy.get_param("pose_theta", 0.0)

            #Time meditions
            self.time = rospy.Time.now()

            #Parameter info for Publishing
            self.msg = Twist()
            self.msg.linear.x = 0
            self.msg.linear.y = 0
            self.msg.linear.z = 0
            self.msg.angular.x = 0
            self.msg.angular.y = 0
            self.msg.angular.z = 0

            #Timer for running the Publisher
            rospy.Timer(rospy.Duration(1.0/50),
self.publish_cmd_vel)

      def get_dt(self):
            current_time = rospy.Time.now()
            dt = (current_time - self.time).to_sec()
            self.time = current_time
            return dt

      #Function that iterates thorugh the list of points and select
each as the current goal
      def set_Trajectory(self, goal: Point):
            self.error_acc_lin = 0.0
            self.error_aac_ang = 0.0

            self.error_prev_lin = 0.0
            self.error_prev_ang = 0.0
            self.local_goal = goal

      def restart_goal(self):
            return self.local_goal == None

      def odom_callback(self, msg):
```

```
            self.odom_pose = msg.pose.pose

      def wrap_to_pi(self, theta):
            result = np.fmod(theta + np.pi, 2 * np.pi)
            if isinstance(theta, np.ndarray):
                  result[result < 0] += 2 * np.pi
            elif result < 0:
                  result += 2 * np.pi
            print(result)
            return result - np.pi

      def compute_error(self):
            error_x = self.local_goal.x - self.odom_pose.position.x
            error_y = self.local_goal.y - self.odom_pose.position.y

         # Distance error
            error_lin = np.sqrt((error_x * error_x) + (error_y *
error_y))
            ref_ang = np.arctan2(error_y, error_x)
         # Angular errror         #theta - Markov
            error_ang = self.wrap_to_pi(ref_ang -
self.odom_pose.orientation.z)


            #Below the Error threshold is considered error-free
            if np.abs(error_lin) < self.lin_tolerance:
                  error_lin = 0.0
            if np.abs(error_ang) < self.ang_tolerance:
                  error_ang = 0.0

            #Publish of the goal (Debugging)
            ref = Twist(linear = Vector3(x = self.local_goal.x, y =
self.local_goal.y, z = 0.0),
                        angular = Vector3(x = 0.0, y = 0.0, z =
ref_ang))
            self.ref_pub.publish(ref)

            #Returning of the linear and angular error
            #(Difference of the robot in actual position and
desired position)
            return error_lin, error_ang
```

```
      def error_derivation(self, error_lin, error_ang):
            error_dev_lin = error_lin - self.error_prev_lin
            error_dev_ang = error_ang - self.error_prev_ang
            self.error_prev_lin = error_lin
            self.error_prev_ang = error_ang
            return error_dev_lin, error_dev_ang

      def publish_cmd_vel(self, _):
            dt = self.get_dt()
            #Components of PID controller
            #Calculation of Proportional Error
            error_lin, error_ang = self.compute_error()
            print_info = "%3f | %3f | %3f | %3f "
%(error_lin,error_ang,self.n_goals,self.index)
            rospy.loginfo(print_info)
            #Calculation of Integral Error
            self.error_int_lin += error_lin * dt
            self.error_int_ang += error_ang * dt

            #Calculation of Derivative Error
            error_dev_lin, error_dev_ang =
self.error_derivation(error_lin, error_ang)

            errorPID = Twist(linear = Vector3(x = error_lin,
                                              y =
self.error_int_lin,
                                             z =
error_dev_lin),

                                    angular = Vector3(x = error_ang,
                                                      y =
self.error_int_ang,
                                                      z =
error_dev_ang))

            twist_msg = Twist(linear = Vector3(x = 0.0, y = 0.0, z
= 0.0),
                                     angular = Vector3(x = 0.0, y =
0.0, z = 0.0))

            if error_ang != 0.0:  #Not alligned directly to the
Goal Coordinate
```

```
                twist_msg.angular.z = (self.kp_ang * error_ang) +
(self.ki_ang * self.error_int_ang) + (self.kd_ang * error_dev_ang)

           elif error_lin != 0.0:      #If its alligned but it
didnt reach the goal yet
                twist_msg.linear.x = (self.kp_lin * error_lin) +
(self.ki_lin * self.error_int_lin) + (self.kd_lin * error_dev_lin)

           else:                       #Goal Reached
                self.index +=1 #Next Point
                self.index = self.index % self.n_goals
                self.set_Trajectory(self.goals[self.index])


           self.cmd_pub.publish(twist_msg)  #Publication of CMD_VEL
topic
           #Publication of Error (Debugging)
           self.error_publisher.publish(errorPID)

#self.lin_tolerance=self.lin_tolerance+self.lin_tolerance

if _name=='main_':

     rospy.init_node("ControlNode")
     CN = ControlNode()

     try:
           rospy.loginfo('The controller node is Running')
```