Proyecto 2 de Robótica

Victoria Medina, Juan Pablo Castillo e Isabel Díaz Miranda

November 29, 2017

Abstract

Se implementó un controlador cinemático de tipo Akermann usando los parámetros de distancia entre los ejes del robot y variables de control a través del sistema operativo ROS y su paquete de simulación "gazebo".

1 Introducción

El objetivo de este proyecto es aplicar los conocimientos adquiridos en clase para implementar un controlador tipo Ackermann. De este modo, aplicar la teoría sobre generación de trayectorias a un problema prático. El controlador se implementó como un nodo de ROS escrito en C++, así como la librería Gazebo.

2 Marco Teórico

2.1 Modelo Ackermann

El modelo que se usa comunmente para un robot de cuatro ruedas es el de "bicicleta". La bicicleta tiene una rueda trasera fijada al cuerpo y el plano de la rueda delantera gira alrededor del eje vertical para guiar al vehículo. La pose del vehículo está representada por el marco de coordenadas V, con su eje x en la dirección de avance del vehículo y su origen en el centro del eje trasero.

La configuración del vehículo está representada por las coordenadas generalizadas q = (x, y, theta) ∈ C donde C ⊂ SE (2). La velocidad del vehículo es, por definición, v en la dirección x del vehículo, y cero en la dirección y, ya que las ruedas no pueden deslizarse lateralmente.

Las líneas punteadas muestran la dirección a lo largo de la cual las ruedas no se pueden mover, las líneas de ningún movimiento, y estas se cruzan en un punto conocido como el Centro Instantáneo de Rotación (CIR). El punto de referencia del vehículo sigue una trayectoria circular y su velocidad angular (figura 2) es y el radio de giro es R1 = L / tan(gamma) donde L es la longitud del vehículo o la base de la rueda. El círculo de giro aumenta con la longitud del vehículo.

Para un ángulo fijo del volante, el automóvil se mueve a lo largo de un arco circular. Hay que notar que R2 > R1 significa que la rueda delantera debe seguir una trayectoria más larga y, por lo tanto, girar más rápido que la rueda trasera. Cuando un vehículo de cuatro ruedas gira alrededor de una esquina, las dos ruedas dirigidas siguen trayectorias circulares de diferentes radios y, por lo

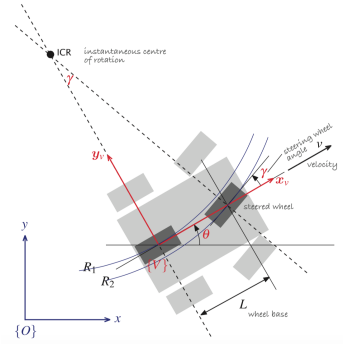

Figure 1: Modelo cinemático de tipo Ackermann.

$$\dot{\theta} = \frac{\nu}{R_1}$$

Figure 2: Fórmula de la velocidad angular.

$$\dot{x} = \nu \cos\theta$$
$$\dot{y} = \nu \sin\theta$$
$$\dot{\theta} = \frac{\nu}{L} \tan\gamma$$

Figure 3: Ecuaciones de movimiento.

tanto, los ángulos de las ruedas directrices gammaL y gammaR deberían ser ligeramente diferentes. Esto se logra con el mecanismo de dirección Ackerman que reduce el desgaste de las llantas. Las ruedas motrices deben girar a diferentes velocidades en las esquinas, por lo que se requiere una caja de cambios diferencial entre el motor y las ruedas motrices.

La velocidad del robot en el marco del mundo es (v cos(theta), v sin (theta)) y combinada con la ecuación de la figura 2 escribimos las ecuaciones del movimiento como la figura 3.

Este modelo se conoce como modelo cinemático ya que describe las velocidades del vehículo pero no las fuerzas o pares que causan la velocidad. La velocidad de cambio del rumbo theta se conoce como velocidad de giro, velocidad de rumbo o velocidad de derrape y se puede medir con un giroscopio.

En el marco de coordenadas del mundo podemos escribir una expresión para la velocidad en la dirección y del vehículo (figura 4) que es la restricción no holonómica. [1]

2.2 ROS

Sistema Operativo Robótico (en inglés Robot Operating System, ROS) es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo.

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu (Linux)) aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows, considerados como 'experimentales'. [2]

$$\dot{y}\cos\theta - \dot{x}\sin\theta \equiv 0$$

Figure 4: Restricción no holonómica.

2.3 Gazebo

Gazebo es un simulador de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma personalizada, crear mundos virtuales usando sencillas herramientas CAD e importar modelos ya creados.

Además, es posible sincronizarlo con ROS de forma que los robots emulados publiquen la información de sus sensores en nodos, así como implementar una lógica y un control que dé ordenes al robot. [3]

3 Implementación

3.1 contoller.cpp

El programa tiene una lista de librerías importadas. El primer método es poseMessageReceived que nos da la posición donde está el robot. Después, mandamos a llamar al Publicador y al Subscriptor y ahí se declaran las variables de la posición a la que se quiere que se mueva el robot. Se usa cout y cin para que el usuario decida estas variables. Luego, se declara el modelo cinemático descargado de github. Finalmente, en el while se declaran las variables del modelo cinemático y se genera el mensaje que se envía a Gazebo.

```
#include <ros/ros.h>
#include <stdlib.h>
#include <iomanip>
#include <geometry_msgs/Pose2D.h>
#include <std_msgs/Float32.h>
#include <math.h>

double x;
double y;
double theta;

using namespace std;
//Metodo que manda llamar la pose
void poseMessageReceived(const geometry_msgs::Pose2D& msg){
    x=msg.x;
    y=msg.y;
    theta=msg.theta;
}

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

    ros::Publisher pub1 = nh.advertise<std_msgs::Float32>("AutoNOMOS_mini/manual_cont
    ros::Publisher pub2 = nh.advertise<std_msgs::Float32>("AutoNOMOS_mini/manual_cont

    double xesperada;
    double yesperada;
    double thetaesperada;

   ros::Subscriber sub = nh.subscribe("robot/pose", 1000, &poseMessageReceived);

    //Pose esperada
    cout<<"X esperada:";
    cin>>xesperada;
    cout<<"Y esperada:";
    cin>>yesperada;
    cout<<"Theta esperada:";
    cin>>thetaesperada;
```

```
    double difx = x-xesperada;
    double dify = y-yesperada;
    double ro = sqrt(difx*difx+dify*dify);
    double alpha = atan2(dify,difx)+theta;
    double beta = -theta-alpha;

    double kro = 1;
    double kalpha = 5;
    double kbeta = -2;
    double v= kro*ro;
    double gamma=kalpha*alpha+kbeta*beta;


    ros::Rate rate(10);


    while(ro>0.08){
        sub = nh.subscribe("robot/pose", 1000, &poseMessageReceived);
        difx=x-xesperada;
        dify=y-yesperada;
        ro=sqrt(difx*difx+dify*dify);
        alpha=atan2(dify,difx)+theta;
        beta=-theta-alpha;
        v=kro*ro;
        gamma=kalpha*alpha+kbeta*beta;

        //Crea mensaje
        std_msgs::Float32 msg1;
        std_msgs::Float32 msg2;
        msg1.data=float(v);
        msg2.data=float(gamma);


        //Publica mensaje.
        pub1.publish(msg1);
        pub2.publish(msg2);

        //Sale del loop
        rate.sleep();
    }
}
```

3.2 CMakeLists.txt

El código utilizado para el CMakeLists se encuentra en el Anexo. Lo modificamos checando que el proyecto que se iba a ejecutar fuera la carpeta donde estaba metido nuestro .cpp. Le cambiamos el find package que es el que necesitamos para poderlo correr y pusimos el mensaje. Se declararon las liberías que ibamos a necesitar para que el controller.cpp se corriera como un ejecutable.

3.3 package.xml

El código creado se encuentra en el anexo. Este solamente tiene roscpp y el catkin para poder correr el contoller.cpp como ejecutable.

3.4 Cómo correr el programa

En primer lugar se descarga el archivo del modelo cinemático de github y se complia para ver que funcionara. En segundo lugar se descarga e instala la librería para ROS llamada Gazebo.

Después, a través de otra terminal se accede a Lentes y se ejecuta el `catkin_make` y luego se ejectuta el source develop setup.bash. Después se corre el `controller_out` usando rosrun, que hace referencia la paquete donde está contenido nuestro código. La terminal le preguna al usuario x y theta esperada y hay que asignarles valores. Hay que confirmar en Gazebo que el robot se está moviendo con la inlinación asignada a theta.

4 Conclusión

Durante este proyecto nos encontramos con varias dificultades. Primero, cuando creamos el primer contoler.cpp hicimos un ciclo infinito en el while que no nos permitía entrar a Gazebo en ningún momento. Pero usando rate.sleep logramos publicar en el Gazebo y se pudo mover el robot.

Otro problema fue que pusimos una lista de librerías con el build y el run depend pero no corría, pero nos dimos cuenta que si solo dejábamos el roscpp y el catkin y con eso instanciar las librerías para poder correr el ejecutable.

Finalmente, logramos ubicar dónde estaban todos los archivos y cómo se conectaban para así lograr eficientemenet que el usuario introdujera datos específicos para el movimiento del robot y verlo en Gazebo.

References

[1] P. Corke, "Robotics, vision and control," *Springer*, pp. 68–69, 2011.

[2] ROS, "About ros," pp. http://www.ros.org/about–ros/, Nov 27, 2017.

[3] ROS, "Gazebo," p. http://wiki.ros.org/gazebo, Nov 27, 2017.

5 Anexo

5.1 Código de CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(controller_aut)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
#find_package(catkin REQUIRED)

## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs std_msgs)

## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
# catkin_python_setup()

################################################
## Declare ROS messages, services and actions ##
################################################
```

```
## To declare and build messages, services or actions from within this
## package, follow these steps:
## * Let MSG_DEP_SET be the set of packages whose message types you use in
##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
## * In the file package.xml:
##   * add a build_depend tag for "message_generation"
##   * add a build_depend and a run_depend tag for each package in MSG_DEP_SET
##   * If MSG_DEP_SET isn't empty the following dependency has been pulled in
##     but can be declared for certainty nonetheless:
##     * add a run_depend tag for "message_runtime"
## * In this file (CMakeLists.txt):
##   * add "message_generation" and every package in MSG_DEP_SET to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * add "message_runtime" and every package in MSG_DEP_SET to
##     catkin_package(CATKIN_DEPENDS ...)
##   * uncomment the add_*_files sections below as needed
##     and list every .msg/.srv/.action file to be processed
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   std_msgs  # Or other packages containing msgs
# )

################################################
## Declare ROS dynamic reconfigure parameters ##
################################################

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
## * In the file package.xml:
##   * add a build_depend and a run_depend tag for "dynamic_reconfigure"
## * In this file (CMakeLists.txt):
##   * add "dynamic_reconfigure" to
##     find_package(catkin REQUIRED COMPONENTS ...)
```

```
##   * uncomment the "generate_dynamic_reconfigure_options" section below
##     and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
#   cfg/DynReconf2.cfg
# )

######################################
## catkin specific configuration ##
######################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES controller_aut
#  CATKIN_DEPENDS other_catkin_pkg
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
 ${catkin_INCLUDE_DIRS}
)

## Declare a C++ library
# add_library(${PROJECT_NAME}
#   src/${PROJECT_NAME}/controller_aut.cpp
# )

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
## either from message generation or dynamic reconfigure
# add_dependencies(${PROJECT_NAME} ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTE

## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
# add_executable(${PROJECT_NAME}_node src/controller_aut_node.cpp)
add_executable(contoller contoller.cpp)
target_link_libraries(contoller ${catkin_LIBRARIES})
#target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})

## Rename C++ executable without prefix
## The above recommended prefix causes long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
```

```
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EX

## Specify libraries to link a library or executable target against
# target_link_libraries(${PROJECT_NAME}_node
#   ${catkin_LIBRARIES}
# )

#############
## Install ##
#############

# all install targets should use catkin DESTINATION variables
# See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html

## Mark executable scripts (Python etc.) for installation
## in contrast to setup.py, you can choose the destination
# install(PROGRAMS
#   scripts/my_python_script
#   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark executables and/or libraries for installation
# install(TARGETS ${PROJECT_NAME} ${PROJECT_NAME}_node
#   ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark cpp header files for installation
# install(DIRECTORY include/${PROJECT_NAME}/
#   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
#   FILES_MATCHING PATTERN "*.h"
#   PATTERN ".svn" EXCLUDE
# )

## Mark other files for installation (e.g. launch and bag files, etc.)
# install(FILES
#   # myfile1
#   # myfile2
#   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
# )

#############
## Testing ##
#############

## Add gtest based cpp test target and link libraries
# catkin_add_gtest(${PROJECT_NAME}-test test/test_controller_aut.cpp)
# if(TARGET ${PROJECT_NAME}-test)
#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# endif()

## Add folders to be run by python nosetests
# catkin_add_nosetests(test)
```

5.2 Código de package.xml

```
<?xml version="1.0"?>
<package format="2">
  <name>controller_aut</name>
  <version>0.0.1</version>
  <description>The controller_aut package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example: -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="victoria@todo.todo">victoria</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/controller_aut</url> -->


  <!-- Author tags are optional, multiple are allowed, one per tag -->
  <!-- Authors do not have to be maintainers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->


  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec dependencies
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use build_export_depend for packages you need in order to build against this pa
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
  <!--   <exec_depend>message_runtime</exec_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
   <buildtool_depend>catkin</buildtool_depend>
  <!--<build_depend>geometry_msgs</build_depend>
  <run_depend>geometry_msgs</run_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>std_msgs</run_depend>
```

```
  <run_depend>roscpp</run_depend>
        -->
        <build_depend>roscpp</build_depend>
</package>
```