Robotic arm chassis linkage control

1. Content Description

This course implements the use of an inverse solution algorithm for the robotic arm to calculate the target pose of the robotic arm and control the robotic arm to move to that pose. At the same time, the chassis also moves synchronously, with the robotic arm and chassis moving in tandem relative to each other.

This section requires entering commands in the terminal. The terminal you open depends on your motherboard type. This lesson uses the Raspberry Pi 5 as an example. For Raspberry Pi and Jetson-Nano boards, you need to open a terminal on the host computer and enter the command to enter the Docker container. Once inside the Docker container, enter the commands mentioned in this section in the terminal. For instructions on entering the Docker container from the host computer, refer to this product tutorial **[Configuration and Operation Guide]--[Enter the Docker (Jetson Nano and Raspberry Pi 5 users, see here)]**.

Simply open the terminal on the Orin motherboard and enter the commands mentioned in this section.

2. Program startup

First, open the terminal and enter the following command to start the robot arm solver.

```
ros2 run arm_kin kin_srv
```

In another terminal, enter

```
ros2 run M3Pro_demo M3Pro_Dancing
```

After the program is started, the robot arm will reach the initial posture. Then enter the following command in the terminal to start it:

```
ros2 topic pub /start_dancing std_msgs/msg/Bool "data: True"  --once
```

After posting this topic, the robot will move forward and the robotic arm will move backward. The robot will move forward 1 meter and then stop. If the above message is posted again, the robot will move back 1 meter.

3. Core code analysis

Code path:

- Raspberry Pi and Jetson-Nano board

  The program code is in the running docker. The path in docker is `/root/yahboomcar_ws/src/M3Pro_demo/M3Pro_demo/ M3Pro_Dancing.py`

- Orin Motherboard

  The program code path is `/home/jetson/yahboomcar_ws/src/M3Pro_demo/M3Pro_demo/ M3Pro_Dancing.py`

Import the used libraries,

```
import rclpy
from std_msgs.msg import Bool
import time
import math
from arm_msgs.msg import ArmJoints
from arm_msgs.msg import ArmJoint
from arm_interface.srv import *
import threading
from rclpy.node import Node
from geometry_msgs.msg import Twist
```

The program initializes and creates publishers and subscribers,

```
def __init__(self, name):
    super().__init__(name)
    self.init_joints = [90, 150, 12, 20, 90, 0]
    #Define the array that stores the current end pose coordinates
    self.CurEndPos =[0.11960304060136814, -0.003155561702528526,
0.3441364762143022, -3.151713610590853e-06, -0.03490854071745721,
-0.028421869689521588]
    #Create a publisher for topics related to six servo controls
    self.pub_SixTargetAngle = self.create_publisher(ArmJoints, "arm6_joints",
10)
    #Create a subscriber to subscribe to the start topic
    self.sub_start =
self.create_subscription(Bool,"/start_dancing",self.startFlagCallBack,100)
    #Create a publisher for the speed topic
    self.CmdVel_pub = self.create_publisher(Twist,"cmd_vel",1)
    #Create a client that calls the robotic arm solution service
    self.client = self.create_client(ArmKinemarics, 'get_kinemarics')
    while not self.client.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('Service not available, waiting again...')
    #Assign the current servo angle value to the servo angle value of the initial
posture
    self.cur_joints = self.init_joints
    #Control the direction variable of the car moving forward and backward
    self.direction = 1
    #Get the current end position of the robotic arm
    self.get_current_end_pos()
    time.sleep(3)
    #CarMotionTime
    self.runtime = 10
    #Car movement speed
    self.vx = 0.1
    self.pubSixArm(self.init_joints)
    print('init done')
```

The startFlagCallBack callback function processes the message data released. If the message is true, two threads are started to control the operation of the robotic arm and chassis respectively. The parameters passed in are self.direction and -self.direction, which indicate the relative direction of movement.

```
def startFlagCallBack(self,msg):
    if msg.data == True:
        base = threading.Thread(target=self.base_move, args=(self.direction,))
        arm = threading.Thread(target=self.arm_move, args=(-self.direction,))
        arm.start()
        base.start()
```

base_move controls the chassis movement function,

```
def base_move(self,base_dir):
    self.vx = self.direction * 0.1
    #Publish movement speed
    self.pubVel(self.vx,0.0,0.0)
    time.sleep(self.runtime)
    #Publish parking speed
    self.pubVel(0.0,0.0,0.0)
    #Convert the value of self.direction, the next run will be in the reverse
direction
    self.direction = -self.direction
```

arm_move controls the movement of the robotic arm.

```
def arm_move(self,arm_dir):
    print("--------------------------------------------------")
    #print("pose_T: ",pose_T)
    #Create service request data
    request = ArmKinemarics.Request()
    #According to the value of arm_dir, increase or decrease in the current x
direction. The amount of increase or decrease is the distance the car moves.
    request.tar_x = self.CurEndPos[0] + arm_dir*(abs(self.vx)*self.runtime)/10
    #y direction remains consistent
    request.tar_y = self.CurEndPos[1]
    #The value in the z direction is calculated by trigonometric function to
obtain the variable, plus the current value
    request.tar_z = math.tan(-0.03490672894389764) * (request.tar_x -
self.CurEndPos[0]) + self.CurEndPos[2]
    print("request.tar_x: ",request.tar_x)
    print("request.tar_z: ",request.tar_z)
    #Call the ik inverse solution to calculate the end-of-arm posture coordinates
from various angles
    request.kin_name = "ik"
    #rpy value should be consistent with the current
    request.roll = self.CurEndPos[3]
    request.pitch  = self.CurEndPos[4]
    request.yaw = self.CurEndPos[5]
    print("calcutelate_request: ",request)
    #The client calls the get_kinemarics service
    future = self.client.call_async(request)
    future.add_done_callback(self.get_ik_respone_callback)
```

get_ik_respone_callback receives the callback function that returns the result of calling the ik service.

```
def get_ik_respone_callback(self, future):
    try:
```

```
            response = future.result()
            joints = [0.0, 0.0, 0.0, 0.0, 0.0,0.0]
            #Assign values to servos 1-3. The assigned values are the values of the
responses returned after the service is processed.
            joints[0] = int(response.joint1) #response.joint1
            joints[1] = int(response.joint2)
            joints[2] = int(response.joint3)
            joints[3] = int(response.joint4)
            joints[4] = 90
            joints[5] = 30
            print("compute_joints: ",joints)
            self.cur_joints = joints
            #Publish a topic about controlling the angles of six servos
            self.pubSixArm(joints)
            time.sleep(1.5)
        except Exception as e:
            self.get_logger().error(f'Service call failed: {e}')
```

get_current_end_pos gets the current end position function of the robotic arm.

```
def get_current_end_pos(self):
    #Create service request data
    request = ArmKinemarics.Request()
    #Assign values to cur_joint1 to cur_joint5 in the request data. The assigned
values are the joint angle values of the current robot arm.
    request.cur_joint1 = float(self.cur_joints[0])
    request.cur_joint2 = float(self.cur_joints[1])
    request.cur_joint3 = float(self.cur_joints[2])
    request.cur_joint4 = float(self.cur_joints[3])
    request.cur_joint5 = float(self.cur_joints[4])
    #Call fk to calculate the end-of-arm posture coordinates from various angles
    request.kin_name = "fk"
    future = self.client.call_async(request)
    future.add_done_callback(self.get_fk_respone_callback)
```

get_fk_respone_callback receives the callback function that returns the result of calling the fk service.

```
def get_fk_respone_callback(self, future):
    try:
        #Receive the result returned after calling the service
        response = future.result()
        #self.get_logger().info(f'Response received: {response.x}')
        #Assign a value to self.CurEndPos. The value assigned is the response
data, including the six values of xyz and rpy.
        self.CurEndPos[0] = response.x
        self.CurEndPos[1] = response.y
        self.CurEndPos[2] = response.z
        self.CurEndPos[3] = response.roll
        self.CurEndPos[4] = response.pitch
        self.CurEndPos[5] = response.yaw
        #self.get_logger().info(f'Response received: {self.CurEndPos}')
        print("self.CurEndPose: ",self.CurEndPos)
    except Exception as e:
        self.get_logger().error(f'Service call failed: {e}')
```