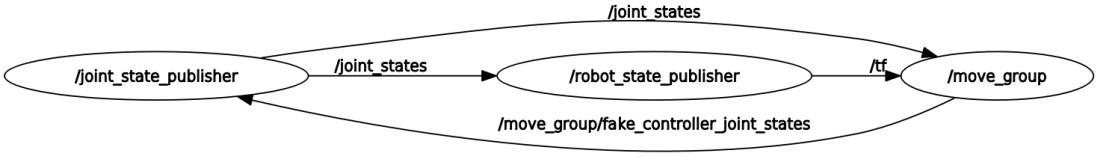
Dynamic Worm-like Motion Planning in Serial Robotic Arms through Inverse Kinematics for Spatial Maneuverability

Introduction

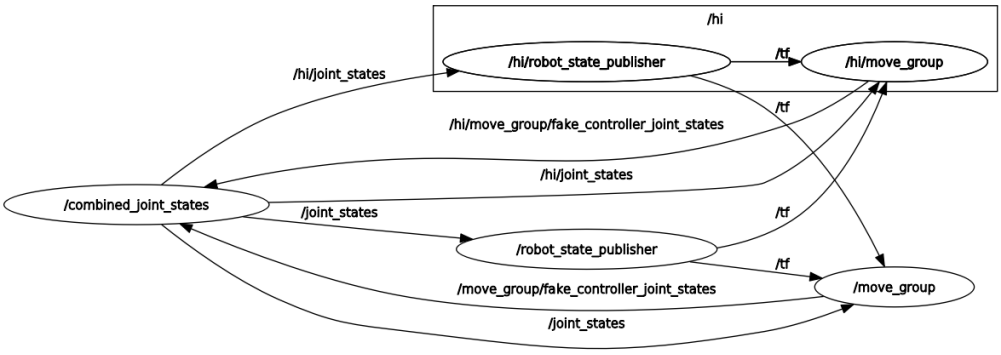
This project unveils a dynamic robotic arm designed to navigate through space utilizing inverse kinematics planning. Departing from the conventional fixed-base model, this arm showcases unparalleled versatility by maneuvering within the spatial environment. Its revolutionary design serves as a pivotal solution, particularly suited for space station operations where agile and adaptable movement capabilities are imperative. Developed and implemented through the Robot Operating System (ROS) and tested in gazebo, this advanced arm technology marks a significant leap forward in the realm of space robotics, offering unprecedented flexibility and precision in spatial exploration and manipulation tasks.

Rviz Visualization

Normally in a robotic arm the base link will be fixed with respect a mobile or a fixed base. The reach of such a robotic arm is fixed and is of limited use especially in a space station. I am using a ros and moveit for doing the same. Moveit doesn't allow planning like inverse part of the planning i.e the base link from the end effector. In order to solve this, i came up an idea of having 2 urdfs, one whose root node is the base link and leaf node as the end effector link and the other with the end effector link as the root node and the base link as the leaf node.Note that the names of joints between the links should be same for this to work. After setting up 2 different configurations for the both urdfs, i made some changes in the demo.launch for rviz visualization which i will come across in the upcoming parts. The normal node graph for moveit looks like below



If i run the move groups under different namespaces , the issue is that both the demo.launch will start its own joint state publisher under different namespaces which publisher to the its own joint state topics. This makes both the groups behave as a different robot, which is not what we want. In order to solve this problem , if i am able to interface joint state publisher to my own custom one , this can be solved. So my joint state publisher (my node is called ‘combined_joint_states’) will take in topic information from both the namespaces of fake controller joint states(/move_group/fake_controller_joint_states and /hi/move_group/fake_controller_joint_states) and publish it to /joint_states and /hi/joint_states which in turn will be taken by both the move groups. So now the node graph will look like



Since the joint names are same it will be really issue to control this , we don't need a difficult inverse mapping to solve this problem

Gazebo engine interface testing

I preferred to gazebo because it has really good compatibility with ros and quite modular and is opensource. Different environments like zero gravity can be tested with the help of it. For gazebo i wrote the hardware interface in my urdfs. I modified the ros_controllers.xml file in the config file for both moveit generated packages like this

```
# MoveIt-specific simulation settings
moveit_sim_hw_interface:
    joint_model_group: controllers_initial_group_
    joint_model_group_pose: controllers_initial_pose_
# Settings for ros_control control loop
generic_hw_control_loop:
    loop_hz: 300
    cycle_time_error_threshold: 0.01
# Settings for ros_control hardware interface
hardware_interface:
  joints:
    - base_link_link_01
    - link_01_link_02
    - link_02_link_03
    - link_03_link_04
    - link_04_link_05
# Creates the /joint_states topic necessary in ROS
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50
arm_controller:
  type: effort_controllers/JointTrajectoryController
  joints:
    - base_link_link_01
    - link_01_link_02
    - link_02_link_03
    - link_03_link_04
    - link_04_link_05
  gains:
```

```
    base_link_link_01:
      p: 1000
      d: 10
      i: 10
      i_clamp: 1
    link_01_link_02:
      p: 1000
      d: 10
      i: 10
      i_clamp: 1
    link_02_link_03:
      p: 5000
      d: 10
      i: 150
      i_clamp: 1
    link_03_link_04:
      p: 1000
      d: 10
      i: 10
      i_clamp: 1
    link_04_link_05:
      p: 1000
      d: 10
      i: 10
      i_clamp: 1

controller_list:
  - name: arm_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: True
    joints:
      - base_link_link_01
      - link_01_link_02
      - link_02_link_03
```

```
      - link_03_link_04
      - link_04_link_05
```

After some bit of pid tuning , the arm is good for launching in the gazebo. Launch the demo_gazebo.launch file for it to work. This part is just the first step. After going through the launch files of both the moveit files. The only necessary statements for it to work is these.

```
<?xml version="1.0"?>
<launch>
  <!-- MoveIt options -->
  <arg name="pipeline" default="ompl" doc="Planning pipeline to

  <!-- Gazebo options -->
  <arg name="gazebo_gui" default="true" doc="Start Gazebo GUI"/>
  <arg name="paused" default="false" doc="Start Gazebo paused"/>
  <arg name="world_name" value="$(find gazebo_ros_link_attacher)
  <arg name="world_pose" default="-x 0 -y 0 -z 1 -R 0 -P 0 -Y 0"

  <!-- Launch Gazebo and spawn the robot -->
  <include file="$(find b47_moveit)/launch/gazebo.launch" pass_a

  <!-- Launch MoveIt -->
  <include file="$(find b47_moveit)/launch/demo.launch" pass_all
    <!-- robot_description is loaded by gazebo.launch, to enable
    <arg name="load_robot_description" value="true" />
    <arg name="moveit_controller_manager" value="ros_control" />
  </include>

  <group ns="inv_arm">
  <!-- <arg name="pipeline" default="ompl" doc="Planning pipelin
  <include file="$(find b47_inv_moveit)/launch/gazebo.launch" pa
  <include file="$(find b47_inv_moveit)/launch/demo.launch" pass
    <!-- robot_description is loaded by gazebo.launch, to enable
    <arg name="load_robot_description" value="true" />
    <arg name="moveit_controller_manager" value="ros_control" />
```

```
  </include>
  </group>
  <node name="joint_publisher_inv" pkg="b47_walking_control" typ

</launch>
```

There is one gazebo gui launcher and two gazebo.launch and demo.launch files for running two parallel move groups under different namespaces. The last node is a custom node which contains

```
#! /usr/bin/env python3
import rospy
from sensor_msgs.msg import JointState
class CommandToJointState:
    def __init__(self):
        self.inv_joint_pub = rospy.Publisher("/inv_arm/joint_sta

        self.gazebo_joint_sub = rospy.Subscriber("/joint_states"
                                                 self.joint_callback,
    def joint_callback(self,msg):
        msg.header.stamp = rospy.Time.now()
        self.inv_joint_pub.publish(msg)

if __name__ == '__main__':
    rospy.init_node('inverted_joint_state_publisher')
    command_to_joint_state = CommandToJointState()
    rospy.spin()
```

which just maps the mimic the joint states of gazebo to another topic /inv_arm/joint_states so that the inverse move group can recognize it. After this running a commander scripts like the below will move the arm

```
#!/usr/bin/env python
import rospy
import moveit_commander
from geometry_msgs.msg import Pose
```

```
import sys
from gazebo_ros_link_attacher.srv import Attach, AttachRequest,

class Move_Arm:
    def __init__(self):
        joint_state_topic = ['joint_states:=/joint_states']
        moveit_commander.roscpp_initialize(joint_state_topic)
        self.robot = moveit_commander.RobotCommander()
        self.group = moveit_commander.MoveGroupCommander("arm")
        moveit_commander.roscpp_initialize(sys.argv)
        self.group.set_pose_reference_frame('base_link')
        pass

    def give_goal(self,p_x,p_y,p_z,x,y,z,w):
        end_effector_link = self.group.get_end_effector_link()
        current_pose = self.group.get_current_pose(end_effector_
        print("End Effector Position:")
        print("X:", current_pose.position.x)
        print("Y:", current_pose.position.y)
        print("Z:", current_pose.position.z)

        print("\nEnd Effector Orientation (Quaternion):")
        print("X:", current_pose.orientation.x)
        print("Y:", current_pose.orientation.y)
        print("Z:", current_pose.orientation.z)
        print("W:", current_pose.orientation.w)

        pose_goal = Pose()
        pose_goal.orientation.w = w
        pose_goal.orientation.x = x
        pose_goal.orientation.y = y
        pose_goal.orientation.z = z
        pose_goal.position.x = p_x
        pose_goal.position.y = p_y
        pose_goal.position.z = p_z
        self.group.set_pose_target(pose_goal)
```

```
        plan = self.group.go(wait=True)
        self.group.stop()
        self.group.clear_pose_targets()

class Attacher:
    def __init__(self):
        self.attach_srv = rospy.ServiceProxy('/link_attacher_no
                                    Attach)
        self.detach_srv = rospy.ServiceProxy('/link_attacher_no
                                    Attach)
        self.attach_srv.wait_for_service()
        self.detach_srv.wait_for_service()

    def attach_request(self,model_name_1,link_name_1,model_name_
        req = AttachRequest()
        req.model_name_1 = model_name_1
        req.link_name_1 = link_name_1
        req.model_name_2 = model_name_2
        req.link_name_2 = link_name_2
        self.attach_srv.call(req)

    def detach_request(self,model_name_1,link_name_1,model_name_
        req = AttachRequest()
        req.model_name_1 = model_name_1
        req.link_name_1 = link_name_1
        req.model_name_2 = model_name_2
        req.link_name_2 = link_name_2
        self.detach_srv.call(req)


if __name__ == "__main__":
    rospy.init_node("move_arm")
    rospy.loginfo("Starting move_arm")
    attacher = Attacher()
    attacher.attach_request("unit_box","link","robot","base_link
```

```
    move_arm = Move_Arm()
    move_arm.give_goal(1.3936,-0.10464,0.2031,0,1,0,0)
```

```
#!/usr/bin/env python
import rospy
import moveit_commander
from geometry_msgs.msg import Pose
import sys
from gazebo_ros_link_attacher.srv import Attach, AttachRequest,

class Move_Arm:
    def __init__(self):
        joint_state_topic = ['joint_states:=/inv_arm/joint_state
        moveit_commander.roscpp_initialize(joint_state_topic)
        self.robot = moveit_commander.RobotCommander()
        self.group = moveit_commander.MoveGroupCommander("inv_ar
        moveit_commander.roscpp_initialize(sys.argv)
        self.group.set_pose_reference_frame('link_05')
        pass

    def give_goal(self,p_x,p_y,p_z,x,y,z,w):
        end_effector_link = self.group.get_end_effector_link()
        current_pose = self.group.get_current_pose(end_effector_
        print("End Effector Position:")
        print("X:", current_pose.position.x)
        print("Y:", current_pose.position.y)
        print("Z:", current_pose.position.z)

        print("\nEnd Effector Orientation (Quaternion):")
        print("X:", current_pose.orientation.x)
        print("Y:", current_pose.orientation.y)
        print("Z:", current_pose.orientation.z)
        print("W:", current_pose.orientation.w)

        pose_goal = Pose()
```

```
        pose_goal.orientation.w = w
        pose_goal.orientation.x = x
        pose_goal.orientation.y = y
        pose_goal.orientation.z = z
        pose_goal.position.x = p_x
        pose_goal.position.y = p_y
        pose_goal.position.z = p_z
        self.group.set_pose_target(pose_goal)
        plan = self.group.go(wait=True)
        self.group.stop()
        self.group.clear_pose_targets()

class Attacher:
    def __init__(self):
        self.attach_srv = rospy.ServiceProxy('/link_attacher_noc
                                    Attach)
        self.detach_srv = rospy.ServiceProxy('/link_attacher_noc
                                    Attach)
        self.attach_srv.wait_for_service()
        self.detach_srv.wait_for_service()

    def attach_request(self,model_name_1,link_name_1,model_name_
        req = AttachRequest()
        req.model_name_1 = model_name_1
        req.link_name_1 = link_name_1
        req.model_name_2 = model_name_2
        req.link_name_2 = link_name_2
        self.attach_srv.call(req)

    def detach_request(self,model_name_1,link_name_1,model_name_
        req = AttachRequest()
        req.model_name_1 = model_name_1
        req.link_name_1 = link_name_1
        req.model_name_2 = model_name_2
        req.link_name_2 = link_name_2
        self.detach_srv.call(req)
```

```
if __name__ == "__main__":
    rospy.init_node("move_arm")
    rospy.loginfo("Starting move_arm")
    attacher = Attacher()
    attacher.detach_request("unit_box","link","robot","base_linl
    attacher.attach_request("unit_box_clone","link","robot","lir
    move_arm = Move_Arm()
    move_arm.give_goal(-1.3936,-0.10464,0.4031,0,1,0,0)
```

The above 2 scripts when run in a different namespaces like for eg.

```
ROS_NAMESPACE=/inv_arm rosrun b47_walking_control move_inv.py
```

This is necessary because the robot description is loaded in this namespace. Inorder to mimic holding on to the ground. I have used an additional dependency of https://github.com/pal-robotics/gazebo_ros_link_attacher which will help in attaching to other links for carrying this out.

A basic demo of this is available in my github

https://github.com/KeerthivasanIITMadras/serial_walking_arm