Project 1 - ROS, Kinematics, and Control

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Introduction to ROS

A. Simple Publisher Subscriber

TASK: Modify the publisher so that instead of "Hello World" it says "Hello CS685!" in addition to the timestamp.

IMAGE: A screen shot from the subscriber.

```
heard hello cs685 1663304677.9917307
heard hello cs685 1663304678.0919507
heard hello cs685 1663304678.1917543
heard hello cs685 1663304678.2919092
heard hello cs685 1663304678.3918223
heard hello cs685 1663304678.4917996
heard hello cs685 1663304678.5918543
heard hello cs685 1663304678.6918938
heard hello cs685 1663304678.791953
heard hello cs685 1663304678.891855
heard hello cs685 1663304678.9918516
heard hello cs685 1663304679.0917513
heard hello cs685 1663304679.191603
heard hello cs685 1663304679.2915316
```

Fig. 1: Simple Publisher and Subscriber

B. Parameters and Launch Files

In this section, we create a launch file to spawn multiple processes with one command; There are some parameters added to such as the name space (group), or node name.

```
hello cs685 1663345415.5555594 [/talker_slow/slow_talker_node]
hello cs685 1663345415.8873546 [/talker_fast/fast_talker_node]
hello cs685 1663345416.2207236 [/talker_fast/fast_talker_node]
hello cs685 1663345416.5540211 [/talker_fast/fast_talker_node]
hello cs685 1663345416.5555825 [/talker_slow/slow_talker_node]
hello cs685 1663345416.8873477 [/talker_fast/fast_talker_node]
hello cs685 1663345417.2206893 [/talker_fast/fast_talker_node]
hello cs685 1663345417.5540514 [/talker_fast/fast_talker_node]
hello cs685 1663345417.5555668 [/talker_slow/slow_talker_node]
```

Fig. 2: Multiple Publishers

From the terminal ran the launch file, the content of published topics are print in Fig. 2. The slow talker topic took 1 second to publish a message, meanwhile the faster one published 3 messages.

Question: As running *rostopic list* on other terminal, there should now be two topics with "chatter" in the name. Their topic names are the same, but their name space (group) are different. One is from the *talker_slow* name space, and the other is from the *talker_fast* one.

C. Subscribing to Multiple Topics

In this section, we create a class that stores messages from multiple topics. In the class, we have two *callback* functions. In the first one, as receiving a message from *talker_fast/chatter*, we save it into a global variable, *message_content*. As getting the message from the *talker_slow/chatter*, we combine its data and the global variable *message_content*, then print the result out and publish it to the *multiple_topics*.

IMAGE: Fig. 3 shows the topics which the listener-class subscribers and publishes. It subscribers two topics: */talker_slow/chatter* and */talker_fast/chatter*, and it publishes a topic: *multiple_topics*.

```
Node [/listener_class]
Publications:
 * /multiple_topics [std_msgs/String]
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /talker_fast/chatter [std_msgs/String]
 * /talker_slow/chatter [std_msgs/String]

Services:
 * /listener_class/get_loggers
 * /listener_class/set_logger_level
```

Fig. 3: A node to listen to multiple topics

IMAGE: Fig. 4 shows the topic's content of the listener-class after combining from the two talkers. The message's content is printed in three rows.

```
hello cs685 1663347921.4321573 [/talker_fast/fast_talker_node]
hello cs685 1663347921.7655199 [/talker_fast/fast_talker_node]
hello cs685 1663347922.0988135 [/talker_fast/fast_talker_node]
hello cs685 1663347922.1042871 [/talker_slow/slow_talker_node]

hello cs685 1663347922.4321537 [/talker_fast/fast_talker_node]
hello cs685 1663347922.7655118 [/talker_fast/fast_talker_node]
hello cs685 1663347923.0988464 [/talker_fast/fast_talker_node]
hello cs685 1663347923.1042788 [/talker_slow/slow_talker_node]
```

Fig. 4: Listener class

II. Kinematics of a Multi-Segment Arm

In this section, we are composing the transformations to represent complex 2D arms and writing a function to support arbitrary numbers of segments.

A. Forward Kinematics of a Multi-Segment Robot

We create a helper function $get\_coords\_from\_arm\_def$ that can takes two lists: (1) $link\_lengths$ a list of lengths of joint angles and (2) $joint\_angles$ a list of angles (in radians) of the arm joints.

A part of the function is shown in Fig. 5. There is a *for* loop which runs through all the joints and calculates the position

```
last_joint_pos = [0, 0, 1]
angle = 0.0
for i in range(len(link_lengths)):
  angle += joint_angles[i]
  T = get_translation(link_lengths[i]) @ get_rot(angle)
  T_inver = np.linalg.inv(T) @ [0,0,1] + last_joint_pos
  all_coords.append(T_inver)
  last_joint_pos = T_inver
```

Fig. 5: The code of the Forward Kinematics function

of the link end point. The link position is accumulated with its previous joint and appended into the *all_coords* list.

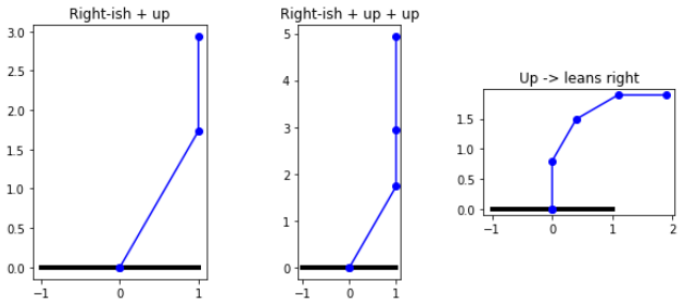


Fig. 6: The arm robot's configurations

The function is then applied to multiple configuration with 2, 3, and 4 segment's arms. The final configurations of the arms are shown in Fig. 6

B. Forward Transforms of a Multi-Segment Robot

In this section, we create an alternate function *get_transform_from_arm_def(link_lengths, joint_angles)* that returns the transform from the endpoint of the robot to the base, so that the resulting SE(3) transform shows both the orientation of the final segment of the arm and the position of the end point.

```
Transform 1 (Right-ish + up)
[
[[0.00 -1.00 0.00]
 [1.00 0.00 1.20]
 [0.00 0.00 1.00]]
[[0.50 -0.87 1.00]
 [0.87 0.50 1.73]
 [0.00 0.00 1.00]]
]
Transform 2(Right-ish+up+up)
[
[[0.00 -1.00 0.00]
 [1.00 0.00 2.00]
 [0.00 0.00 1.00]]
[[0.00 -1.00 0.00]
 [1.00 0.00 1.20]
 [0.00 0.00 1.00]]
[[0.50 -0.87 1.00]
 [0.87 0.50 1.73]
 [0.00 0.00 1.00]]
]
```

```
Transform 3(Up->leansright)
[
[[1.00 -0.00 0.80]
 [0.00 1.00 0.00]
 [0.00 0.00 1.00]]
[[0.87 -0.50 0.69]
 [0.50 0.87 0.40]
 [0.00 0.00 1.00]]
[[0.50 -0.87 0.40]
 [0.87 0.50 0.69]
 [0.00 0.00 1.00]]
[[0.00 -1.00 0.00]
 [1.00 0.00 0.80]
 [0.00 0.00 1.00]]
]
```

Fig. 7: Transformation from the end segment to the base

In Fig. 7, the transformations of three arms configuration are shown. For example in *Transform 1*, the last segment is rotated $\pi/2$, and its y is 1.2 unit higher than the previous segment. The first segment is rotated $\pi/3$, and the endpoint is at position $(1.0, 1.73)$.

C. Inverse kinematics of a two-segment arm

Implement a function $inverse\_kinematics\_two_segment\_arm(link\_le$ that computes the inverse kinematic solution for a two-segment robot arm.

```
l1, l2 = link_lengths
x, y = position
dis_to_goal_2 = np.square(x) + np.square(y)
phi = np.arccos((dis_to_goal_2 + np.square(l1)-np.square(l2))/
                (2*l1*np.sqrt(dis_to_goal_2)))
b = np.sqrt(dis_to_goal_2)*np.sin(phi)
if dis_to_goal_2 > np.square(l1) + np.square(l2):
  theta2 = np.pi - np.arcsin(b/l2)
else:
  theta2 = np.arcsin(b/l2)

if x == 0.0 and y > 0.0:
  beta = np.pi/2
elif x == 0.0 and y < 0.0:
  beta = -np.pi/2
elif y ==0.0 and x > 0.0:
  beta = 0.0
elif y == 0.0 and x < 0.0:
  beta = np.pi
else:
  beta = np.arctan2(y,x)
angle2 = theta2 - np.pi
if theta2 > np.pi:
  angle1 = beta - phi
else:
  angle1 = beta + phi
return angle1, angle2 # theta_1, theta_2
```

Fig. 8: The code for IK of 2 segments' arm

To do the calculation, we assume the end point of the second segment reach the goal, and we create a triangle between the origin, the endpoint of the first segment and the goal. On the triangle, we use the sin and $cosin$ laws to calculate the angle ϕ and θ_2. From the two angles, we can determine the angle β. There are some special positions which need to treat differently if they are the goals. The code detail is shown in Fig. 8. As running the code for several arm configurations, we have the results in Fig. 9.

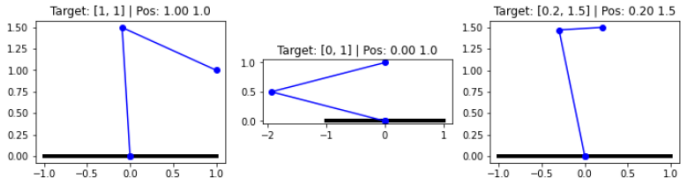


Fig. 9: The figure for IK of 2 segments' arm

D. Inverse Kinematics for a three-segment arm

In this section, we want to solve for the joint angles of a three-segment arm. Not only do we want to put the end at a particular point, but we also want to control the angle of the final segment of the arm.

The code is shown in Fig. 10, and quite straightforward. From the desired angle of the last segment, we can calculate the end point of the second arm segment, and it should be fixed. This point is also the goal of the first two segments. Therefore, we call the function $inverse\_kinematics\_two\_segment\_arm$ with the length of

segment 1 and segment 2, and the desire target for the endpoint of the second segment. The angle of the first joint is the different between the desire angle and the two first joint angles.

```
delta_xl3 = l3*np.cos(angle)
delta_yl3 = l3*np.sin(angle)
new_x = x - delta_xl3
new_y = y - delta_yl3
angle1, angle2 = inverse_kinematics_two_segment_arm([l1,l2], [new_x, new_y])
angle3 = angle - angle1 - angle2
return angle1, angle2, angle3 #theta_1, theta_2, theta_3
```

Fig. 10: The code of IK for 3 segments arm

Use the new inverse kinematics function, we can determine the joint's angle for one target or follow trajectories as shown in Fig. 11 and 12.

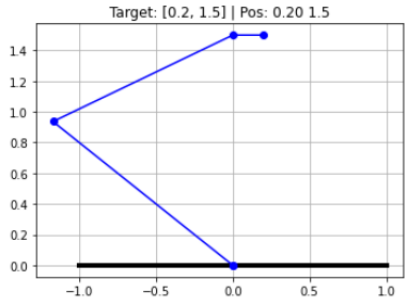


Fig. 11: One point

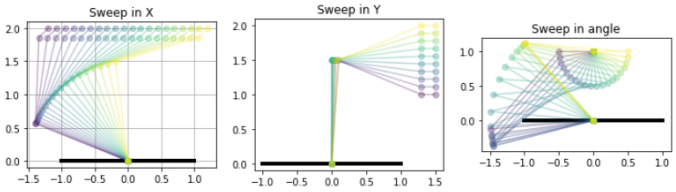
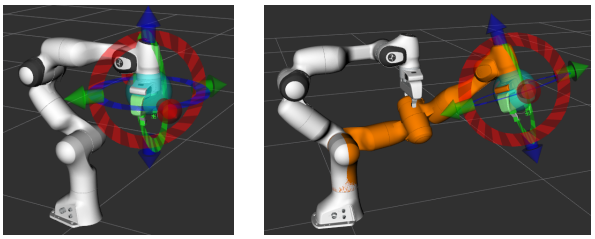


Fig. 12: 3 Trajectories

III. Experimenting with MoveIt ROS

In this problem, I simply work through one of the basic tutorials for the MoveIt motion planning and simulation framework.

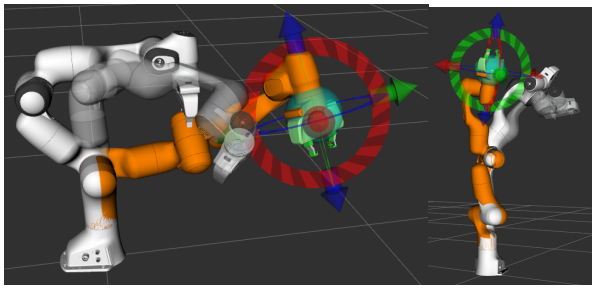


(a) Initial Position (b) Set up the goal

Fig. 13: Set up the initial and target position

In Fig. 13a), it is the initial position of the Panda manipulator. In Fig. 13b), we select a random valid pose as the target for the manipulator (in orange color).

Fig. 14a shows the planning step to generate a feasible trajectory for the manipulator move from the initial position



(a) Motion Planning and Executing (b) Another example

Fig. 14: Motion Planning

to the goal. To execute the trajectory, we press the button *execute* on the MoveIt. Fig. 14b shows another case for the manipulator.

IV. A Differential Drive Vehicle

A. A simple differential drive controller

TASK Complete the controller for this simple Differential Drive Vehicle. The vehicle should simultaneously turn and move (so it should not turn in place and then move; it should instead form smooth trajectories to the target point). (1) It should turn at a rate proportional to the relative angle between the vehicle and the angle to the goal (theta_rel). (2) The speed of the vehicle should be proportional to the distance to the goal.

Define the Vehicle Velocities: The vehicle is actuated by two wheels with v_l and v_r, which, as consequence, generates the rotational velocity ω.

$$\omega = \frac{v_r - r_l}{l} \tag{1}$$

We calculate the vehicle linear velocity at the center of mass:

$$vel = \omega R = \frac{v_r - r_l}{l} \frac{l}{2} \frac{v_l + v_r}{v_r - v_l} = \frac{v_l + v_r}{2} \tag{2}$$

Combine two equations above, so the velocity of the left and right wheels are:

$$v_l = vel - 0.5\omega l$$
$$v_r = vel + 0.5\omega l$$

```
if traj:
  if dist < 0.4:
    break
vl = None
vr = None
if vl is None or vr is None:
    omega1 = omega_vel_ratio*(K_theta*theta_rel + \
                        0.01*(theta_rel-last_theta_rel)/dt)
    vel1 = K_vel*dist + 0.02*(dist-last_dist)
    vl = vel1 - 0.5*omega1*car.width
    vr = vel1 + 0.5*omega1*car.width
last_theta_rel = theta_rel
last_dist = dist
```

Fig. 15: Simple controller for the vehicle

Using the equation above to calculate the *left* and *right wheels' velocities* shown in Fig. 15. The *traj* condition will

be used the late section. Here, we use a PD controller to calculate the ω and vehicle linear velocity vel. To let the vehicle make the sharply, we add a factor $omega\_vel\_ratio$ to the ω formula to increase the influence of the rotation.

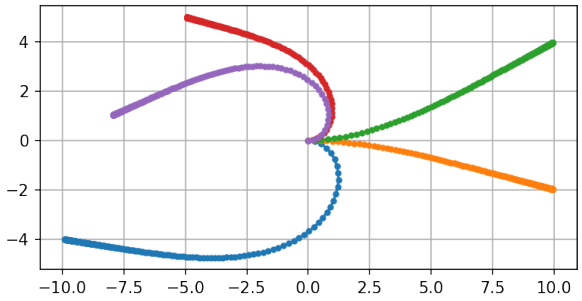


Fig. 16: The vehicle's trajectories

Using the controller, we can steer the vehicle to go to five required goals, and the result is shown in Fig. 16.

B. A more complex controller

To use acceleration or deceleration to control the vehicle to its goal, we need to change the model of differential drive vehicle.

1) *The internal state parameters*: we need to define the *max torque*, wheel's radius, and rotation velocities of the left right wheels: ω_r and ω_l
2) *The command function*: we need to change its inputs from $velocities$ to $torque$: τ_l and τ_r. The outputs are the angular velocities of the left and right wheels ω_r and ω_l.

```
def command(self, to_l, to_r):
    if np.abs(to_l) > self.max_torque:
        to_l = to_l / np.abs(to_l)
    if np.abs(to_r) > self.max_torque:
        to_r = to_r / np.abs(to_r)
    self.tau_l = to_l
    self.tau_r = to_r
```

Fig. 17: New command function

3) *The advance_time function*: We need to convert the torques τ_r and τ_l into the angular velocities ω_r and ω_l, and then linear velocities of the wheels: v_r and v_l.

```
self.omega_r += self.tau_r*dt
self.omega_l += self.tau_l*dt
self.vr = self.omega_r*self.r
self.vl = self.omega_l*self.r
```

Fig. 18: Additional code for advance-time function

PID Control: The PID control should similar to Fig. 19.

C. Waypoint following

TASK: write a new function that follows a series of waypoints. The robot should navigate towards a waypoint and then, when it gets within a distance of 0.5 of the current waypoint, it should switch to the next.

```
if to_l is None or to_r is None:
    I_theta += theta_rel*dt
    I_dist = dist*dt
    to_rot = K_theta_p*theta_rel + K_theta_d*(theta_rel-last_theta_rel)/dt + 0.05*I_theta
    to_linear = K_vel_p*dist + K_vel_d*(last_dist-dist)/dt + 0.05*I_dist
    to_l = (to_linear - to_rot/car.width)*0.5*car.r
    to_r = (to_linear + to_rot/car.width)*0.5*car.r
last_dist = dist
last_theta_rel = theta_rel
```

Fig. 19: PID control

To ease the new function in this section, we add a *traj* condition in the simple-controller which it is triggered by this function. The condition will break the *for* loop of the controller if the vehicle moves close to the goal as 0.5 unit; then we follow the next way point. The code to let the vehicle to follow the waypoints are quite simple. It consists a *for* loop running from the first waypoint to the last one. For each way point, it will call the function *simple_diff_drive_controller* and the *traj* condition is set to **True**.

```
1  def follow_a_trajectory(car, trajectory):
2    for i in range(len(trajectory)):
3      simple_diff_drive_controller(car, trajectory[i][0], \
4          trajectory[i][1], K_theta=0.025, K_vel=0.05, traj=True)
```

```
1  plt.figure(figsize=(6, 6), dpi=150)
2  plt.grid()
3  car = DiffDrive()
4  trajectory = [[1,5], [-5,3], [-5,-3], [4,10]]
5  follow_a_trajectory(car, trajectory)
6  car.visualize_trajectory()
```

Fig. 20: The code to follow multiple waypoints

The vehicle's trajectory to follow the waypoints is shown in Fig. 21.

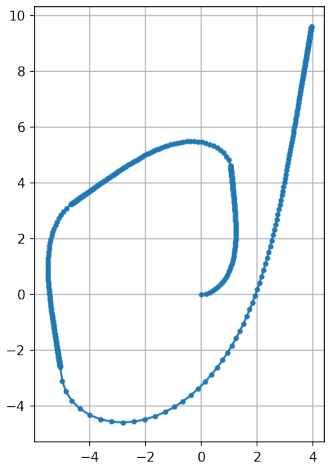


Fig. 21: Follow multiple waypoints