To implement leader-follower formation control using the turtlesim simulator, we can follow the following steps:

Launch the turtlesim simulator: Open a terminal window and type the command "roscore" to start the ROS master. Then, open another terminal window and type the command "rosrun turtlesim turtlesim\_node" to start the turtlesim node.

Spawn multiple turtles: We can use the "rosrun turtlesim turtle\_teleop\_key" command to spawn a new turtle in the turtlesim simulator. We can repeat this command to spawn multiple turtles. We can also give each turtle a unique name.

Designate a leader turtle: We can designate a leader turtle by selecting one of the spawned turtles. The leader turtle can follow an arbitrary trajectory by using the "rosrun turtlesim turtle\_teleop\_key" command or by writing a custom ROS node to control its motion.

Implement follower behavior: The followers should keep track of the leader turtle's position and orientation and maintain a desired distance and orientation relative to the leader turtle. We can use ROS messages to communicate between the leader and follower turtles. The leader turtle can publish its position and orientation to a ROS topic, and the follower turtles can subscribe to this topic and adjust their position and orientation accordingly.

Write a custom ROS node: We can write a custom ROS node to implement the follower behavior. The node can subscribe to the leader turtle's position and orientation topic and use this information to calculate the desired position and orientation of the follower turtle. The node can then publish velocity commands to the follower turtle to move it to the desired position and orientation.

Test the implementation: We can test the implementation by running the custom ROS node and observing the behavior of the leader and follower turtles in the turtlesim simulator. We can adjust the parameters of the follower behavior to achieve different formation shapes and sizes.

Overall, the implementation of leader-follower formation control using the turtlesim simulator involves spawning multiple turtles, designating a leader turtle, implementing follower behavior using ROS messages and a custom ROS node, and testing the implementation in the turtlesim simulator.

To implement leader-follower formation control using Python and ROS, we can follow these steps:

Create a ROS package: Open a terminal window and type the command "mkdir leader\_follower" to create a directory for the ROS package. Then, type the command "cd leader\_follower" to navigate to the directory. Finally, type the command "catkin\_create\_pkg leader\_follower rospy turtlesim" to create a new ROS package with the necessary dependencies.

Create a Python script for the leader turtle: In the "src" directory of the "leader\_follower" package, create a new Python script called "leader.py". This script will control the motion of the leader turtle. Import the necessary ROS packages and create a publisher to send velocity commands to the leader turtle. Then, write a loop that publishes the desired velocity commands based on the leader's trajectory. Finally, add the necessary ROS boilerplate code to initialize the ROS node and spin the loop.

Here's an example code snippet for the "leader.py" script:

#!/usr/bin/env python

import rospy

from geometry\_msgs.msg import Twist

def main():

rospy.init\_node('leader')

pub = rospy.Publisher('/turtle1/cmd\_vel', Twist, queue\_size=10)

rate = rospy.Rate(10)

while not rospy.is\_shutdown():

# Calculate the desired velocity based on the leader's trajectory

vel = Twist()

vel.linear.x = 1.0

vel.angular.z = 0.5

pub.publish(vel)

rate.sleep()

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except rospy.ROSInterruptException:

pass

Create a Python script for the follower turtles: In the "src" directory of the "leader\_follower" package, create a new Python script called "follower.py". This script will control the motion of the follower turtles. Import the necessary ROS packages and create a subscriber to receive the leader turtle's position and orientation. Then, write a loop that calculates the desired position and orientation for the follower turtles based on the leader's position and orientation. Finally, add the necessary ROS boilerplate code to initialize the ROS node and spin the loop.

Here's an example code snippet for the "follower.py" script:  
#!/usr/bin/env python

import rospy

from geometry\_msgs.msg import Twist, PoseStamped

from math import atan2, sqrt

leader\_pose = PoseStamped()

def leader\_callback(data):

global leader\_pose

leader\_pose = data

def main():

rospy.init\_node('follower')

pub = rospy.Publisher('/turtle2/cmd\_vel', Twist, queue\_size=10)

sub = rospy.Subscriber('/turtle1/pose', PoseStamped, leader\_callback)

rate = rospy.Rate(10)

while not rospy.is\_shutdown():

# Calculate the desired position and orientation based on the leader's position and orientation

follower\_pose = PoseStamped()

follower\_pose.pose.position.x = leader\_pose.pose.position.x + 2.0

follower\_pose.pose.position.y = leader\_pose.pose.position.y

follower\_pose.pose.orientation.z = leader\_pose.pose.orientation.z

follower\_pose.pose.orientation.w = leader\_pose.pose.orientation.w

vel = Twist()

vel.linear.x = sqrt((follower\_pose.pose.position.x - leader\_pose.pose.position.x)\*\*2 + (follower\_pose.pose.position.y - leader\_pose.pose.position.y)\*\*2)

vel.angular.z = atan2(follower\_pose.pose.position.y - leader\_pose.pose.position.y, follower\_pose.pose.position.x - leader\_pose.pose.position.x) - leader\_pose.pose.orientation.z

pub.publish(vel)

rate.sleep()

if \_\_name\_\_ == '\_\_main\_\_':

try:

To implement this behavior, we need to modify the "leader.py" script to detect when the followers are in the required formation position and then turn to a random direction before moving in a straight line.

Here's an example code snippet that implements this behavior:

#!/usr/bin/env python

import rospy

from geometry\_msgs.msg import Twist, PoseStamped

from math import atan2, sqrt

import random

leader\_pose = PoseStamped()

follower1\_pose = PoseStamped()

follower2\_pose = PoseStamped()

def leader\_callback(data):

global leader\_pose

leader\_pose = data

def follower1\_callback(data):

global follower1\_pose

follower1\_pose = data

def follower2\_callback(data):

global follower2\_pose

follower2\_pose = data

def main():

rospy.init\_node('leader')

leader\_pub = rospy.Publisher('/turtle1/cmd\_vel', Twist, queue\_size=10)

follower1\_sub = rospy.Subscriber('/turtle2/pose', PoseStamped, follower1\_callback)

follower2\_sub = rospy.Subscriber('/turtle3/pose', PoseStamped, follower2\_callback)

rate = rospy.Rate(10)

while not rospy.is\_shutdown():

# Check if the followers are in the required formation position

formation\_distance = 2.0

if sqrt((follower1\_pose.pose.position.x - leader\_pose.pose.position.x)\*\*2 + (follower1\_pose.pose.position.y - leader\_pose.pose.position.y)\*\*2) < formation\_distance and sqrt((follower2\_pose.pose.position.x - leader\_pose.pose.position.x)\*\*2 + (follower2\_pose.pose.position.y - leader\_pose.pose.position.y)\*\*2) < formation\_distance:

# Turn to a random direction

vel = Twist()

vel.linear.x = 0.0

vel.angular.z = random.uniform(-1.0, 1.0)

leader\_pub.publish(vel)

# Keep moving in a straight line

vel.linear.x = 1.0

vel.angular.z = 0.0

leader\_pub.publish(vel)

else:

# Calculate the desired velocity based on the leader's trajectory

vel = Twist()

vel.linear.x = 1.0

vel.angular.z = 0.5

leader\_pub.publish(vel)

rate.sleep()

if \_\_name\_\_ == '\_\_main\_\_':

try:

main()

except rospy.ROSInterruptException:

pass

In this code, we added two new subscriber callbacks for the follower turtles, and we modified the main loop to check if the followers are in the required formation position. If they are, the leader turtle turns to a random direction by setting its angular velocity to a random value between -1.0 and 1.0. Then, it keeps moving in a straight line by setting its linear velocity to 1.0 and its angular velocity to 0.0. Otherwise, it continues moving along its trajectory with a linear velocity of 1.0 and an angular velocity of 0.5.

To implement this behavior, we need to modify the "leader.py" script to detect when the leader turtle approaches to the border and then send an instruction to the followers to teleport them back to their initial position. Then, the leader turtle needs to return to the center using the custom service defined in "pid\_server.py". If the PID server fails to move the leader turtle to the goal location, it should be teleported back to the center. Finally, all turtles should be moved back to their initial positions with a black pen.

Here's an example code snippet that implements this behavior:

#!/usr/bin/env python

import rospy

from geometry\_msgs.msg import Twist, PoseStamped

from math import atan2, sqrt

import random

from turtlesim.srv import TeleportAbsolute

from my\_custom\_srv\_msg\_pkg.srv import MyCustomServiceMessage, MyCustomServiceMessageResponse

leader\_pose = PoseStamped()

follower1\_pose = PoseStamped()

follower2\_pose = PoseStamped()

def leader\_callback(data):

global leader\_pose

leader\_pose = data

def follower1\_callback(data):

global follower1\_pose

follower1\_pose = data

def follower2\_callback(data):

global follower2\_pose

follower2\_pose = data

def teleport\_followers():

# Teleport follower1 to its initial position

rospy.wait\_for\_service('/turtle2/teleport\_absolute')

try:

teleport = rospy.ServiceProxy('/turtle2/teleport\_absolute', TeleportAbsolute)

teleport(4.0, 2.0, 0.0)

except rospy.ServiceException as e:

print("Service call failed: %s"%e)

# Teleport follower2 to its initial position

rospy.wait\_for\_service('/turtle3/teleport\_absolute')

try:

teleport = rospy.ServiceProxy('/turtle3/teleport\_absolute', TeleportAbsolute)

teleport(4.0, 8.0, 0.0)

except rospy.ServiceException as e:

print("Service call failed: %s"%e)

def move\_to\_center():

rospy.wait\_for\_service('move\_turtle\_to\_goal')

try:

move\_turtle = rospy.ServiceProxy('move\_turtle\_to\_goal', MyCustomServiceMessage)

resp1 = move\_turtle(5.544, 5.544)

if resp1.success == True:

print("Leader turtle moved to the center successfully")

else:

print("Failed to move the leader turtle to the center")

# Teleport the leader turtle back to the center

rospy.wait\_for\_service('/turtle1/teleport\_absolute')

try:

teleport = rospy.ServiceProxy('/turtle1/teleport\_absolute', TeleportAbsolute)

teleport(5.544, 5.544, 0.0)

except rospy.ServiceException as e:

print("Service call failed: %s"%e)

except rospy.ServiceException as e:

print("Service call failed: %s"%e)

def main():

rospy.init\_node('leader')

leader\_pub = rospy.Publisher('/turtle1/cmd\_vel', Twist, queue\_size=10)

follower1\_sub = rospy.Subscriber('/turtle2/pose', PoseStamped, follower1\_callback)

follower2\_sub = rospy.Subscriber('/turtle3/pose', PoseStamped, follower2\_callback)

rate = rospy.Rate(10)

while not rospy.is\_shutdown():

# Check if the leader turtle approaches to the border

border\_distance = 0.5

if leader\_pose.pose.position.x < border\_distance or leader\_pose.pose.position.x > 10 - border\_distance or leader\_pose.pose.position.y < border\_distance or leader\_pose.pose.position.y > 10 -