Dingo folder

Dingo:

1. Launch
    a. Dingo.launch
    b. Dingo_gazebo_sim.launch
    c. dingo_simulator.launch

2. Scripts
    a. Dingo_driver.py
    b. run_robot.py

3. src/dingo
    a. _init_.py
    b. Status_publisher.py

4. CMakeLists.txt

5. Package.xml

6. setup.py

1. launch:

1. a) dingo.launch:

<launch> 🟦 Launch file begins 🟦

<arg name="is_sim" default="0"/> 🟦 Argument: is_sim = 0 means simulation is disabled by default 🟦
<arg name="is_physical" default="1"/> 🟦 Argument: is_physical = 1 means physical hardware is enabled 🟦
<arg name="use_joystick" default="1"/> 🟦 Argument: use_joystick = 1 enables joystick control 🟦
<arg name="use_keyboard" default="0"/> 🟦 Argument: use_keyboard = 0 disables keyboard control 🟦
<arg name="serial_port" default="/dev/ttyS0"/> 🟦 Argument: serial port path for communication with Arduino 🟦
<arg name="use_imu" default="0"/> 🟦 Argument: IMU usage disabled by default 🟦

<group if="$(arg is_physical)"> 🟦 Begin group: only launches if physical mode is enabled 🟦

<node pkg="rosserial_python" type="serial_node.py" name="dingo_rosserial" args="$(arg serial_port)" output="screen"/> 🟦 Node to handle serial communication with Arduino 🟦
<node pkg="dingo_peripheral_interfacing" type="dingo_lcd_interfacing.py" name="dingo_LCD_node" output="screen"></node> 🟦 LCD interfacing node to show messages or data 🟦
</group> 🟦 End group for physical robot configuration 🟦

<group if="$(arg use_joystick)"> 🟦 Begin group: only launches if joystick is enabled 🟦
<node pkg="joy" type="joy_node" name="JOYSTICK"> 🟦 Joystick node to read input from game controller 🟦
<param name="autorepeat_rate" value="30"/> 🟦 Sends repeated joystick commands at 30Hz 🟦
<!-- <param name="joy_node/dev" value="/dev/input/js0"/> --> 🟦 Optional device parameter for joystick (commented out) 🟦
<!-- <arg name="coalesce_interval" value="0.02"/> --> 🟦 Optional timing adjustment for input coalescing (commented out) 🟦
</node> 🟦 End joystick node 🟦
</group> 🟦 End group for joystick input 🟦

<group if="$(arg use_keyboard)"> 🟦 Begin group: only launches if keyboard input is enabled 🟦
<node pkg="dingo_input_interfacing" type="Keyboard.py" name="keyboard_input_listener" output="screen"></node> 🟦 Keyboard input listener node 🟦
</group> 🟦 End group for keyboard input 🟦

<node pkg="dingo" type="dingo_driver.py" name="dingo" args="$(arg is_sim) $(arg is_physical) $(arg use_imu)" output="screen"/> 🟦 Main driver node for the Dingo robot, takes simulation, physical, and IMU flags 🟦

</launch> 🟦 End of launch file 🟦

1. b) dingo_gazebo_sim.launch:

<launch> 🟦 Start of launch file 🟦
<include file="$(find dingo_gazebo)/launch/simulation.launch" /> 🟦 Include the simulation launch file from the dingo_gazebo package 🟦
</launch> 🟦 End of launch file 🟦

1. c) dingo_simulator.launch:

<launch> 🟦 Start of launch file 🟦
  <include file="$(find dingo_gazebo)/launch/simulation.launch" /> 🟦 Include the simulation launch file from the dingo_gazebo package 🟦
</launch> 🟦 End of launch file 🟦

2. scripts:

2. a) dingo_driver.py:

import numpy as np 🟦 Import NumPy for numerical and array operations 🟦
import time 🟦 Import time module for timing functions 🟦
import rospy 🟦 Import rospy for ROS Python client library 🟦
import sys 🟦 Import sys module to access command-line arguments 🟦
from std_msgs.msg import Float64 🟦 Import Float64 message type from standard ROS messages 🟦
import signal 🟦 Import signal module to handle system signals like interrupts 🟦
import socket 🟦 Import socket module (not used in this snippet, potentially for networking) 🟦
import platform 🟦 Import platform module (not used here, can check OS info) 🟦
from dingo_peripheral_interfacing.msg import ElectricalMeasurements 🟦 Import custom ROS message for electrical sensor data 🟦

# Fetching is_sim and is_physical from command-line arguments 🟦
args = rospy.myargv(argv=sys.argv) 🟦 Get ROS-compatible argument list 🟦
if len(args) != 4: # arguments have not been provided, go to defaults (not sim, is physical) 🟦
  is_sim = 0 🟦 Default: not simulation mode 🟦
  is_physical = 1 🟦 Default: physical robot mode enabled 🟦
  use_imu = 1 🟦 Default: IMU sensor enabled 🟦
else: 🟦 If exactly 3 arguments are passed after the script name 🟦
  is_sim = int(args[1]) 🟦 Parse simulation flag from argument 🟦
  is_physical = int(args[2]) 🟦 Parse physical robot flag from argument 🟦
  use_imu = int(args[3]) 🟦 Parse IMU usage flag from argument 🟦

from dingo_control.Controller import Controller 🟦 Import main controller class for robot control 🟦
from dingo_input_interfacing.InputInterface import InputInterface 🟦 Import class to handle user joystick input 🟦
from dingo_control.State import State, BehaviorState 🟦 Import robot state and behavior enums/classes 🟦

from dingo_control.Kinematics import four_legs_inverse_kinematics 🟦 Import inverse kinematics function for quadruped legs 🟦
from dingo_control.Config import Configuration 🟦 Import configuration parameters for robot 🟦
from dingo_control.msg import TaskSpace, JointSpace, Angle 🟦 Import custom ROS message types for control commands 🟦
from std_msgs.msg import Bool 🟦 Import Bool message type for emergency stop status 🟦

if is_physical: 🟦 If running on physical hardware, import hardware interfaces 🟦
    from dingo_servo_interfacing.HardwareInterface import HardwareInterface 🟦 Interface to servo hardware 🟦
    from dingo_peripheral_interfacing.IMU import IMU 🟦 IMU sensor interface 🟦
    from dingo_control.Config import Leg_linkage 🟦 Mechanical leg linkage configuration 🟦

class DingoDriver: 🟦 Main driver class for controlling the Dingo robot 🟦
    def __init__(self, is_sim, is_physical, use_imu): 🟦 Constructor with mode flags passed in 🟦
        self.message_rate = 50 🟦 Control loop frequency in Hz 🟦
        self.rate = rospy.Rate(self.message_rate) 🟦 ROS Rate object to maintain loop timing 🟦

        self.is_sim = is_sim 🟦 Save simulation mode flag 🟦
        self.is_physical = is_physical 🟦 Save physical hardware flag 🟦
        self.use_imu = use_imu 🟦 Save IMU usage flag 🟦

        self.joint_command_sub = rospy.Subscriber("/joint_space_cmd", JointSpace, self.run_joint_space_command) 🟦 Subscribe to joint space command topic 🟦
        self.task_command_sub = rospy.Subscriber("/task_space_cmd", TaskSpace, self.run_task_space_command) 🟦 Subscribe to task space command topic 🟦
        self.estop_status_sub = rospy.Subscriber("/emergency_stop_status", Bool, self.update_emergency_stop_status) 🟦 Subscribe to emergency stop status topic 🟦
        self.external_commands_enabled = 0 🟦 Flag indicating whether external control commands are accepted 🟦

if self.is_sim: 🟦 Check if running in simulation mode 🟦
    self.sim_command_topics = [🟦 List of ROS topics to publish joint commands in simulation 🟦
        "/dingo_controller/FR_theta1/command", 🟦 Front Right leg, joint 1 command topic 🟦
        "/dingo_controller/FR_theta2/command", 🟦 Front Right leg, joint 2 command topic 🟦

"/dingo_controller/FR_theta3/command", 🟦 Front Right leg, joint 3 command topic 🟦
"/dingo_controller/FL_theta1/command", 🟦 Front Left leg, joint 1 command topic 🟦
"/dingo_controller/FL_theta2/command", 🟦 Front Left leg, joint 2 command topic 🟦
"/dingo_controller/FL_theta3/command", 🟦 Front Left leg, joint 3 command topic 🟦
"/dingo_controller/RR_theta1/command", 🟦 Rear Right leg, joint 1 command topic 🟦
"/dingo_controller/RR_theta2/command", 🟦 Rear Right leg, joint 2 command topic 🟦
"/dingo_controller/RR_theta3/command", 🟦 Rear Right leg, joint 3 command topic 🟦
"/dingo_controller/RL_theta1/command", 🟦 Rear Left leg, joint 1 command topic 🟦
"/dingo_controller/RL_theta2/command", 🟦 Rear Left leg, joint 2 command topic 🟦
"/dingo_controller/RL_theta3/command" 🟦 Rear Left leg, joint 3 command topic 🟦
]

self.sim_publisher_array = [] 🟦 Initialize empty list to hold publishers for simulation command topics 🟦
for i in range(len(self.sim_command_topics)): 🟦 Iterate over each simulation command topic 🟦
self.sim_publisher_array.append(rospy.Publisher(self.sim_command_topics[i], Float64, queue_size=0)) 🟦 Create a ROS publisher for each joint command topic with Float64 message type, append to list 🟦

# Create robot configuration instance 🟦
self.config = Configuration() 🟦 Instantiate Configuration object which holds robot parameters 🟦

if is_physical: 🟦 If running on physical hardware 🟦
self.linkage = Leg_linkage(self.config) 🟦 Create leg linkage object using configuration (mechanical parameters) 🟦
self.hardware_interface = HardwareInterface(self.linkage) 🟦 Create hardware interface to control servos via linkage model 🟦

if self.use_imu: 🟦 If IMU sensor usage enabled 🟦
self.imu = IMU() 🟦 Instantiate IMU interface object to read sensor data 🟦

Create controller and user input handles 🟦

```
self.controller = Controller( 🟦 Instantiate main robot controller 🟦
    self.config,  🟦 Pass robot configuration 🟦
    four_legs_inverse_kinematics,  🟦 Pass inverse kinematics function for controlling leg joints 🟦
)

self.state = State() 🟦 Initialize the robot state object 🟦
rospy.loginfo("Creating input listener...") 🟦 Log info about input listener creation 🟦
self.input_interface = InputInterface(self.config) 🟦 Initialize input interface with config 🟦
rospy.loginfo("Input listener successfully initialised... Robot will now receive commands via Joy messages") 🟦 Log successful initialization 🟦

rospy.loginfo("Summary of current gait parameters:") 🟦 Log summary heading 🟦
rospy.loginfo("overlap time: %.2f", self.config.overlap_time) 🟦 Log overlap time parameter 🟦
rospy.loginfo("swing time: %.2f", self.config.swing_time) 🟦 Log swing time parameter 🟦
rospy.loginfo("z clearance: %.2f", self.config.z_clearance) 🟦 Log z clearance parameter 🟦
rospy.loginfo("back leg x shift: %.2f", self.config.rear_leg_x_shift) 🟦 Log back leg x shift parameter 🟦
rospy.loginfo("front leg x shift: %.2f", self.config.front_leg_x_shift) 🟦 Log front leg x shift parameter 🟦


def run(self): 🟦 Main run loop for robot control 🟦
    while not rospy.is_shutdown(): 🟦 Continue loop until ROS shutdown signal 🟦
        if self.state.currently_estopped == 1: 🟦 Check if emergency stop is active 🟦
            rospy.logwarn("E-stop pressed. Controlling code now disabled until E-stop is released") 🟦 Warn about e-stop activation 🟦
            self.state.trotting_active = 0 🟦 Disable trotting gait 🟦
            while self.state.currently_estopped == 1: 🟦 Wait while e-stop remains active 🟦
                self.rate.sleep() 🟦 Sleep to maintain loop rate 🟦
            rospy.loginfo("E-stop released") 🟦 Log e-stop release event 🟦

        rospy.loginfo("Manual robot control active. Currently not accepting external commands") 🟦 Log that manual control is active 🟦
        command = self.input_interface.get_command(self.state,self.message_rate) 🟦 Get joystick command from input interface 🟦
        self.state.behavior_state = BehaviorState.REST 🟦 Set robot behavior state to REST 🟦
        self.controller.run(self.state, command) 🟦 Run controller update with current state and command 🟦
```

```
        self.controller.publish_joint_space_command(self.state.joint_angles) 🟦 Publish joint angles to ROS topics 🟦
        self.controller.publish_task_space_command(self.state.rotated_foot_locations) 🟦 Publish foot positions in task space 🟦
        if self.is_sim: 🟦 If running in simulation mode 🟦
            self.publish_joints_to_sim(self.state.joint_angles) 🟦 Publish joint commands to simulation topics 🟦
        if self.is_physical: 🟦 If running on physical robot hardware 🟦
            self.hardware_interface.set_actuator_postions(self.state.joint_angles) 🟦 Update servo actuator positions with joint angles 🟦
        while self.state.currently_estopped == 0: 🟦 Loop while e-stop is not active 🟦
            time.start = rospy.Time.now() 🟦 Capture current ROS time 🟦

            command = self.input_interface.get_command(self.state,self.message_rate) 🟦 Get updated joystick command 🟦
            if command.joystick_control_event == 1: 🟦 Check if joystick requested external control 🟦
                if self.state.currently_estopped == 0: 🟦 Confirm e-stop is not active 🟦
                    self.external_commands_enabled = 1 🟦 Enable external control commands 🟦
                    break 🟦 Exit inner loop to switch control mode 🟦
                else: 🟦 If e-stop active when external control requested 🟦
                    rospy.logerr("Received Request to enable external control, but e-stop is pressed so the request has been ignored. Please release e-stop and try again") 🟦 Log error and ignore request 🟦

            self.state.euler_orientation = ( 🟦 Update IMU orientation if available, else zero vector 🟦
                self.imu.read_orientation() if self.use_imu else np.array([0, 0, 0])
            )
            [yaw,pitch,roll] = self.state.euler_orientation 🟦 Unpack orientation into yaw, pitch, roll 🟦

            self.controller.run(self.state, command) 🟦 Step the controller forward using latest state and command 🟦

if self.state.behavior_state == BehaviorState.TROT or self.state.behavior_state == BehaviorState.REST: 🟦 Check if robot is in TROT or REST behavior 🟦
    self.controller.publish_joint_space_command(self.state.joint_angles) 🟦 Publish current joint angles 🟦
    self.controller.publish_task_space_command(self.state.rotated_foot_locations) 🟦 Publish current foot positions in task space 🟦
    # rospy.loginfo(state.joint_angles) 🟦 (commented out) Log joint angles 🟦
```

    # rospy.loginfo('State.height: ', state.height) 🟦 (commented out) Log robot height state 🟦

    if self.is_sim: 🟦 If running in simulation mode 🟦
        self.publish_joints_to_sim(self.state.joint_angles) 🟦 Publish joint commands to the simulator 🟦
    if self.is_physical: 🟦 If running on physical robot 🟦
        self.hardware_interface.set_actuator_postions(self.state.joint_angles) 🟦 Update PWM signals to servos 🟦

    # rospy.loginfo('All angles: \n',np.round(np.degrees(state.joint_angles),2)) 🟦 (commented out) Log all joint angles in degrees 🟦
    time.end = rospy.Time.now() 🟦 Record end time for control loop iteration 🟦
    # rospy.loginfo(str(time.start-time.end)) 🟦 (commented out) Log time taken for control iteration 🟦

    # rospy.loginfo('State: \n',state) 🟦 (commented out) Log entire robot state 🟦
else: 🟦 If behavior state is not TROT or REST 🟦
    if self.is_sim: 🟦 If in simulation mode 🟦
        self.publish_joints_to_sim(self.state.joint_angles) 🟦 Publish joints to simulator regardless of behavior 🟦
self.rate.sleep() 🟦 Sleep to maintain loop rate 🟦

if self.state.currently_estopped == 0: 🟦 If emergency stop is not active 🟦
    rospy.loginfo("Manual Control deactivated. Now accepting external commands") 🟦 Log manual control deactivation 🟦
    command = self.input_interface.get_command(self.state,self.message_rate) 🟦 Get input command from interface 🟦
    self.state.behavior_state = BehaviorState.REST 🟦 Set behavior state to REST 🟦
    self.controller.run(self.state, command) 🟦 Run controller update with new command 🟦
    self.controller.publish_joint_space_command(self.state.joint_angles) 🟦 Publish joint angles to ROS topics 🟦
    self.controller.publish_task_space_command(self.state.rotated_foot_locations) 🟦 Publish foot locations in task space 🟦
    if self.is_sim: 🟦 If in simulation 🟦
        self.publish_joints_to_sim(self.state.joint_angles) 🟦 Publish joint commands to sim 🟦
    if self.is_physical: 🟦 If on physical robot 🟦
        self.hardware_interface.set_actuator_postions(self.state.joint_angles) 🟦 Update servo PWM widths 🟦
    while self.state.currently_estopped == 0: 🟦 While no emergency stop 🟦
        command = self.input_interface.get_command(self.state,self.message_rate) 🟦 Get updated input command 🟦

```
        if command.joystick_control_event == 1: 🟦 If joystick requested manual control 🟦
            self.external_commands_enabled = 0 🟦 Disable external commands (enable
manual control) 🟦
            break 🟦 Exit this control loop 🟦
        self.rate.sleep() 🟦 Sleep to maintain loop rate 🟦

def update_emergency_stop_status(self, msg): 🟦 Callback to update emergency stop
status 🟦
    if msg.data == 1: 🟦 If e-stop is pressed 🟦
        self.state.currently_estopped = 1 🟦 Set e-stop active flag 🟦
    if msg.data == 0: 🟦 If e-stop is released 🟦
        self.state.currently_estopped = 0 🟦 Clear e-stop flag 🟦
    return 🟦 End function 🟦

def run_task_space_command(self, msg): 🟦 Run commands in task space (foot
positions) 🟦
    if self.external_commands_enabled == 1 and self.currently_estopped == 0: 🟦 If
external control enabled and no e-stop 🟦
        foot_locations = np.zeros((3,4)) 🟦 Initialize foot location array (3 coords, 4 legs) 🟦
        j = 0 🟦 Initialize index for coordinate component 🟦
        for i in 3: 🟦 Iterate over 3 coordinates (BUG: should be `range(3)`) 🟦
            foot_locations[i] = [msg.FR_foot[j], msg.FL_foot[j], msg.RR_foot[j],
msg.RL_foot[j]] 🟦 Assign foot positions for each leg 🟦
            j = j+1 🟦 Increment coordinate index 🟦
        print(foot_locations) 🟦 Print foot location matrix for debugging 🟦
        joint_angles = self.controller.inverse_kinematics(foot_locations, self.config) 🟦
Calculate joint angles from foot locations 🟦
        if self.is_sim: 🟦 If simulation mode 🟦
            self.publish_joints_to_sim(self, joint_angles) 🟦 Publish calculated joint angles to
sim (BUG: extra self) 🟦

        if self.is_physical: 🟦 If physical robot 🟦
            self.hardware_interface.set_actuator_postions(joint_angles) 🟦 Send joint
commands to hardware servos 🟦

    elif self.external_commands_enabled == 0: 🟦 If external control is disabled 🟦
        rospy.logerr("ERROR: Robot not accepting commands. Please deactivate manual
control before sending control commands") 🟦 Log error for ignored command 🟦
    elif self.currently_estopped == 1: 🟦 If e-stop active 🟦
        rospy.logerr("ERROR: Robot currently estopped. Please release before trying to
send commands") 🟦 Log error for e-stop state 🟦

def run_joint_space_command(self, msg): 🟦 Run commands in joint space (direct joint
angles) 🟦
```

```
    if self.external_commands_enabled == 1 and self.currently_estopped == 0: 🟦 If external control enabled and no e-stop 🟦
        joint_angles = np.zeros((3,4)) 🟦 Initialize joint angles array 🟦
        j = 0 🟦 Initialize coordinate index 🟦
        for i in 3: 🟦 Iterate over 3 coordinates (BUG: should be `range(3)`) 🟦
            joint_angles[i] = [msg.FR_foot[j], msg.FL_foot[j], msg.RR_foot[j], msg.RL_foot[j]] 🟦 Assign joint angles from message 🟦
            j = j+1 🟦 Increment index 🟦
        print(joint_angles) 🟦 Print joint angles for debugging 🟦

        if self.is_sim: 🟦 If simulation 🟦
            self.publish_joints_to_sim(self, joint_angles) 🟦 Publish joint commands to sim (BUG: extra self) 🟦

        if self.is_physical: 🟦 If physical robot 🟦
            self.hardware_interface.set_actuator_postions(joint_angles) 🟦 Send commands to hardware 🟦

    elif self.external_commands_enabled == 0: 🟦 If external control disabled 🟦
        rospy.logerr("ERROR: Robot not accepting commands. Please deactivate manual control before sending control commands") 🟦 Log error 🟦
    elif self.currently_estopped == 1: 🟦 If e-stop active 🟦
        rospy.logerr("ERROR: Robot currently estopped. Please release before trying to send commands") 🟦 Log error 🟦

def publish_joints_to_sim(self, joint_angles): 🟦 Publish joint angles to simulation topics 🟦
    rows, cols = joint_angles.shape 🟦 Get shape of joint angles matrix 🟦
    i = 0 🟦 Initialize publisher index 🟦
    for col in range(cols): 🟦 Iterate over columns (legs) 🟦
        for row in range(rows): 🟦 Iterate over rows (joints) 🟦
            self.sim_publisher_array[i].publish(joint_angles[row,col]) 🟦 Publish angle to simulator topic 🟦
            i = i + 1 🟦 Increment publisher index 🟦


def signal_handler(sig, frame): 🟦 Handle system signals for clean exit 🟦
    sys.exit(0) 🟦 Exit program immediately 🟦

def main(): 🟦 Main program entry point 🟦
    """Main program
    """ 🟦 Docstring 🟦
    rospy.init_node("dingo_driver") 🟦 Initialize ROS node named 'dingo_driver' 🟦
```

signal.signal(signal.SIGINT, signal_handler) 🟦 Register Ctrl-C handler for graceful shutdown 🟦

dingo = DingoDriver(is_sim, is_physical, use_imu) 🟦 Create an instance of DingoDriver with simulation, physical, and IMU flags 🟦
dingo.run() 🟦 Start running the main control loop of the robot driver 🟦

main() 🟦 Call the main function to start the program 🟦

2. b) run_robot.py:

import numpy as np 🟦 Import NumPy library for numerical operations 🟦
import time 🟦 Import time module to manage time-related tasks 🟦
import rospy 🟦 Import ROS Python client library 🟦
import sys 🟦 Import sys module to access system-specific parameters and functions 🟦
from std_msgs.msg import Float64 🟦 Import ROS standard message type Float64 🟦
import signal 🟦 Import signal module to handle interrupts and termination signals 🟦
import socket 🟦 Import socket module for network communication 🟦
import platform 🟦 Import platform module to access system information 🟦
from dingo_peripheral_interfacing.msg import ElectricalMeasurements 🟦 Import custom ROS message for electrical measurements 🟦
import subprocess 🟦 Import subprocess module to run system commands (used for I2C tests) 🟦

args = rospy.myargv(argv=sys.argv) 🟦 Get command-line arguments passed to the ROS node 🟦
if len(args) != 3: 🟦 Check if arguments were not provided (expecting 2 extra args) 🟦
    is_sim = 0 🟦 Default: simulation mode is off 🟦
    is_physical = 0 🟦 Default: physical hardware is off 🟦
else: 🟦 Else, arguments are provided 🟦
    is_sim = int(args[1]) 🟦 Set simulation mode based on the second command-line argument 🟦
    is_physical = int(args[2]) 🟦 Set physical hardware mode based on the third command-line argument 🟦

from dingo_peripheral_interfacing.IMU import IMU 🟦 Import IMU module to read orientation data 🟦
from dingo_control.Controller import Controller 🟦 Import main robot controller module 🟦
from dingo_input_interfacing.InputInterface import InputInterface 🟦 Import module that handles external input controls 🟦
from dingo_control.State import State 🟦 Import State class to keep track of robot's state 🟦

```
from dingo_control.Kinematics import four_legs_inverse_kinematics 🟦 Import function to calculate leg joint angles 🟦
from dingo_control.Config import Configuration 🟦 Import configuration parameters for the robot 🟦
from dingo_input_interfacing.InputController import InputController 🟦 Import controller for handling user input 🟦

if is_physical: 🟦 Only import hardware-specific modules if running on real robot 🟦
    from dingo_servo_interfacing.HardwareInterface import HardwareInterface 🟦 Import interface to control hardware servos 🟦
    from dingo_control.Config import Leg_linkage 🟦 Import leg linkage configuration data 🟦

def signal_handler(sig, frame): 🟦 Define handler for termination signal (e.g., Ctrl+C) 🟦
    sys.exit(0) 🟦 Exit the program safely 🟦

def main(use_imu=False): 🟦 Define the main function with optional IMU usage 🟦
    """Main program""" 🟦 Docstring describing the main program 🟦
    rospy.init_node("dingo") 🟦 Initialize the ROS node with the name "dingo" 🟦
    message_rate = 50 🟦 Set the rate at which messages will be processed (50 Hz) 🟦
    rate = rospy.Rate(message_rate) 🟦 Create a Rate object to sleep to maintain 50 Hz 🟦

    signal.signal(signal.SIGINT, signal_handler) 🟦 Register signal handler to allow clean shutdown with Ctrl+C 🟦

    #TODO: Create a publisher for joint states 🟦 Placeholder comment for publishing joint states 🟦
    if is_sim: 🟦 Check if simulation mode is enabled 🟦
        command_topics = [ 🟦 List of ROS topics to command each joint in Gazebo 🟦
            "/notspot_controller/FR1_joint/command", 🟦 Front Right leg joint 1 🟦
            "/notspot_controller/FR2_joint/command", 🟦 Front Right leg joint 2 🟦
            "/notspot_controller/FR3_joint/command", 🟦 Front Right leg joint 3 🟦
            "/notspot_controller/FL1_joint/command", 🟦 Front Left leg joint 1 🟦
            "/notspot_controller/FL2_joint/command", 🟦 Front Left leg joint 2 🟦
            "/notspot_controller/FL3_joint/command", 🟦 Front Left leg joint 3 🟦
            "/notspot_controller/RR1_joint/command", 🟦 Rear Right leg joint 1 🟦
            "/notspot_controller/RR2_joint/command", 🟦 Rear Right leg joint 2 🟦
            "/notspot_controller/RR3_joint/command", 🟦 Rear Right leg joint 3 🟦
            "/notspot_controller/RL1_joint/command", 🟦 Rear Left leg joint 1 🟦
            "/notspot_controller/RL2_joint/command", 🟦 Rear Left leg joint 2 🟦
            "/notspot_controller/RL3_joint/command" 🟦 Rear Left leg joint 3 🟦
        ]
```

```
        publishers = [] 🟦 List to store publishers for each joint command topic 🟦
        for i in range(len(command_topics)): 🟦 Loop through each topic and create a
publisher 🟦
            publishers.append(rospy.Publisher(command_topics[i], Float64, queue_size = 0))
🟦 Create and store publisher 🟦

    config = Configuration() 🟦 Create robot configuration object 🟦
    if is_physical: 🟦 If running on physical hardware 🟦
        linkage = Leg_linkage(config) 🟦 Create leg linkage configuration 🟦
        hardware_interface = HardwareInterface(linkage) 🟦 Create interface to control
hardware servos 🟦
        if use_imu: 🟦 If IMU is enabled 🟦
            imu = IMU(port="/dev/ttyACM0") 🟦 Create IMU object using the specified port
🟦
            imu.flush_buffer() 🟦 Clear any old IMU data 🟦

    controller = Controller( 🟦 Create controller object for gait control 🟦
        config, 🟦 Pass configuration to controller 🟦
        four_legs_inverse_kinematics, 🟦 Pass kinematics function 🟦
    )
    state = State() 🟦 Create initial robot state object 🟦
    print("Creating input listener...") 🟦 Notify user about input listener creation 🟦
    input_interface = InputInterface(config) 🟦 Create input interface object 🟦
    print(platform.processor()) 🟦 Print processor information (useful for platform-specific
actions) 🟦
    input_Controller = InputController(1, platform.processor()) 🟦 Create input controller
object with ID and processor info 🟦
    print("Done.") 🟦 Confirm setup is complete 🟦

    last_loop = time.time() 🟦 Record time of last loop iteration 🟦

    print("Summary of gait parameters:") 🟦 Print header for gait parameter summary 🟦
    print("overlap time: ", config.overlap_time) 🟦 Display configured overlap time 🟦
    print("swing time: ", config.swing_time) 🟦 Display configured swing time 🟦
    print("z clearance: ", config.z_clearance) 🟦 Display leg height clearance 🟦
    print("x shift: ", config.x_shift) 🟦 Display horizontal shift value 🟦

    loop = 0 🟦 Initialize loop counter 🟦
    while not rospy.is_shutdown(): 🟦 Run while ROS is not shut down 🟦
        print("Waiting for L1 to activate robot.") 🟦 Wait for user to press activation button
🟦

        while True: 🟦 Loop until activation event is received 🟦
```

```
            command = input_interface.get_command(state,message_rate) 🟦 Get user command from input device 🟦
            if command.joystick_control_event == 1: 🟦 If activation button is pressed 🟦
                break 🟦 Exit the inner loop 🟦
            rate.sleep() 🟦 Sleep to maintain loop rate 🟦

        print("Robot activated.") 🟦 Notify that robot is now active 🟦

        while True: 🟦 Main control loop 🟦
            time.start = rospy.Time.now() 🟦 Record start time of loop iteration 🟦
            command = input_interface.get_command(state,message_rate) 🟦 Get latest input command 🟦
            if command.joystick_control_event == 1: 🟦 If deactivation button is pressed 🟦
                print("Deactivating Robot") 🟦 Print deactivation message 🟦
                break 🟦 Exit the control loop 🟦

            quat_orientation = ( 🟦 Read orientation from IMU or set default if IMU is not used 🟦
                imu.read_orientation() if use_imu else np.array([1, 0, 0, 0]) 🟦 Read or assign default quaternion 🟦
            )
            state.quat_orientation = quat_orientation 🟦 Update robot state with orientation 🟦

            controller.run(state, command) 🟦 Run the controller to update joint angles 🟦

            if is_sim: 🟦 If running in simulation 🟦
                rows, cols = state.joint_angles.shape 🟦 Get shape of joint angle matrix 🟦
                print(rows) 🟦 Print number of rows (legs) 🟦
                print(cols) 🟦 Print number of columns (joints per leg) 🟦
                for row in range(rows): 🟦 Loop through each leg 🟦
                    for col in range(cols): 🟦 Loop through each joint in leg 🟦
                        publishers[rows*row+col].publish(state.joint_angles[row, col]) 🟦 Publish joint angle to Gazebo 🟦

            if is_physical: 🟦 If running on physical robot 🟦
                hardware_interface.set_actuator_postions(state.joint_angles) 🟦 Send joint angles to servos 🟦

            time.end = rospy.Time.now() 🟦 Record end time of loop iteration 🟦
            loop +=1 🟦 Increment loop counter 🟦
            rate.sleep() 🟦 Sleep to maintain loop rate 🟦

main() 🟦 Call the main function to start the program 🟦
```

3. src/dingo:

3. b) status_publisher.py:

import rospy 🟦 Import the ROS Python client library 🟦
from std_msgs.msg import String 🟦 Import the String message type from std_msgs 🟦

class StatusPublisher: 🟦 Define a class to handle publishing robot status messages 🟦

    def __init__(self): 🟦 Constructor method called when an object of StatusPublisher is created 🟦
        self.status_publisher = rospy.Publisher("/robot_status_messages", String, queue_size = 10) 🟦 Create a ROS publisher on the '/robot_status_messages' topic with message type 'String' and a queue size of 10 🟦

    def publish_message(self, message): 🟦 Define a method to publish a string message 🟦
        self.status_publisher.publish(message) 🟦 Publish the given message to the '/robot_status_messages' topic 🟦

4. CMakeLists.txt:

cmake_minimum_required(VERSION 3.0.2) 🟦 Minimum required CMake version is 3.0.2 🟦
project(dingo) 🟦 Define the name of the project as 'dingo' 🟦

find_package(catkin REQUIRED COMPONENTS 🟦 Find and load catkin with the required components 🟦
  rospy 🟦 rospy is needed for writing ROS nodes in Python 🟦
)

catkin_python_setup() 🟦 Setup Python package so ROS can recognize and install it properly 🟦

catkin_package(🟦 Define a catkin package (optional fields commented below) 🟦
INCLUDE_DIRS include 🟦 Uncomment and set if there are header files in 'include' folder 🟦
LIBRARIES dingo 🟦 Uncomment if building C++ libraries in this package 🟦
CATKIN_DEPENDS rospy 🟦 Uncomment if other catkin packages are needed at build/runtime 🟦
DEPENDS system_lib 🟦 Uncomment if non-catkin dependencies are needed 🟦
)

include_directories(🟦 Specify directories to be included when compiling 🟦
include 🟦 Uncomment if you have header files in 'include' folder 🟦
  ${catkin_INCLUDE_DIRS} 🟦 Include directories from catkin packages 🟦
)

catkin_install_python(PROGRAMS 🟦 Install Python scripts so they can be used as executables 🟦
   scripts/run_robot.py 🟦 Install the run_robot.py script 🟦
   scripts/dingo_driver.py 🟦 Install the dingo_driver.py script 🟦
   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}) 🟦 Install them to the standard catkin binary directory 🟦

5. Package.xml:

<?xml version="1.0"?> 🟦 XML declaration; required at the top of the file 🟦
<package format="2"> 🟦 Declares the ROS package format version (2 is current) 🟦

  <name>dingo</name> 🟦 Name of the ROS package 🟦
  <version>0.0.0</version> 🟦 Package version (update with changes/releases) 🟦
  <description>The dingo package</description> 🟦 Short description of the package 🟦

  <maintainer email="alex@todo.todo">alex</maintainer> 🟦 Package maintainer and contact email 🟦
  <license>TODO</license> 🟦 License type (e.g., MIT, BSD, Apache-2.0) — replace TODO 🟦

  <buildtool_depend>catkin</buildtool_depend> 🟦 Specifies catkin as the build tool 🟦

  <build_depend>rospy</build_depend> 🟦 rospy needed at build time 🟦
  <build_export_depend>rospy</build_export_depend> 🟦 Needed for building other packages that depend on this one 🟦
  <exec_depend>rospy</exec_depend> 🟦 Needed at runtime for this package to function 🟦

  <depend>numpy</depend> 🟦 Python NumPy library is used by this package 🟦
  <depend>time</depend> 🟦 'time' is a built-in Python module — this line is unnecessary 🟦

  <export> 🟦 Optional: place extra export information for tools or build systems here 🟦
    <!-- Other tools can request additional information be placed here -->
  </export>

</package>

6. setup. py:

from distutils.core import setup 🟦 Imports the setup function to handle package installation 🟦
from catkin_pkg.python_setup import generate_distutils_setup 🟦 Imports catkin helper to generate setup config for ROS 🟦

d = generate_distutils_setup(🟦 Generates a dictionary with package info for setup() 🟦
    packages=['dingo'], 🟦 Lists the Python package(s) in this ROS package (must match folder name in src/) 🟦
    package_dir={'': 'src'} 🟦 Maps root ('') to 'src' directory where the Python code is located 🟦
)

setup(**d) 🟦 Runs setup() with the generated config to register the ROS Python package 🟦