Dingo Control

/msg folder

The `msg/` folder inside `dingo_control` defines custom ROS messages used for communication between different parts of the robot's control system. These messages are crucial for describing the robot's motion both in terms of desired joint angles and desired foot positions in 3D space.

Files inside `msg/`:

1. `Angle.msg`

float32 theta1

float32 theta2

float32 theta3

Code Explanation:

- theta1: First angle (can represent the base or hip joint rotation).
- theta2: Second angle (can represent the upper leg or thigh movement).
- theta3: Third angle (can represent the lower leg or knee movement).

All angles are expressed in radians for consistency in robotics calculations.

Why Declared This Way?

- The robot's motion depends on controlling multiple joints—each joint needs a separate angle.
- Using separate fields (theta1, theta2, theta3) allows precise individual control of each joint.
- Declaring as float32 keeps the message lightweight and efficient for real-time robotic applications.
- This format is especially useful in inverse kinematics, where specific joint angles are calculated and sent to actuators.

Algorithm:

1. Decide how many joints or angles you need (3 in this case).

2. Choose the data type: float32 is precise and lightweight.

3. Name each field meaningfully: theta1, theta2, theta3.

4. Save it as a .msg file for use in ROS messages.

Flowchart:

2. `JointSpace.msg`

How it is Created?

`JointSpace.msg` defines the joint angles for each of the four legs individually.

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id

dingo_control/Angle FL_foot
dingo_control/Angle FR_foot
dingo_control/Angle RL_foot
dingo_control/Angle RR_foot
```

Code Explanation:

- **Header**: Standard ROS header for time and frame tracking.
- **FL_foot, FR_foot, RL_foot, RR_foot**: Each is a custom `Angle` message containing specific joint angles (hip, upper_leg, lower_leg) for the respective leg.

This message is **published by the gait controller** (e.g., `trot_controller.py`) and **read by the motor driver** to control the angles directly.

Why Declared This Way?

- **Joint angles** provide low-level control over the robot.
- Directly setting angles ensures **precise, quick actuation**.
- Dividing by leg (FL, FR, RL, RR) allows **independent control**.
- Using a `Header` ensures **timing and frame consistency**.

Algorithm:

Step 1: Initialize Header

- Set `seq` = previous seq + 1

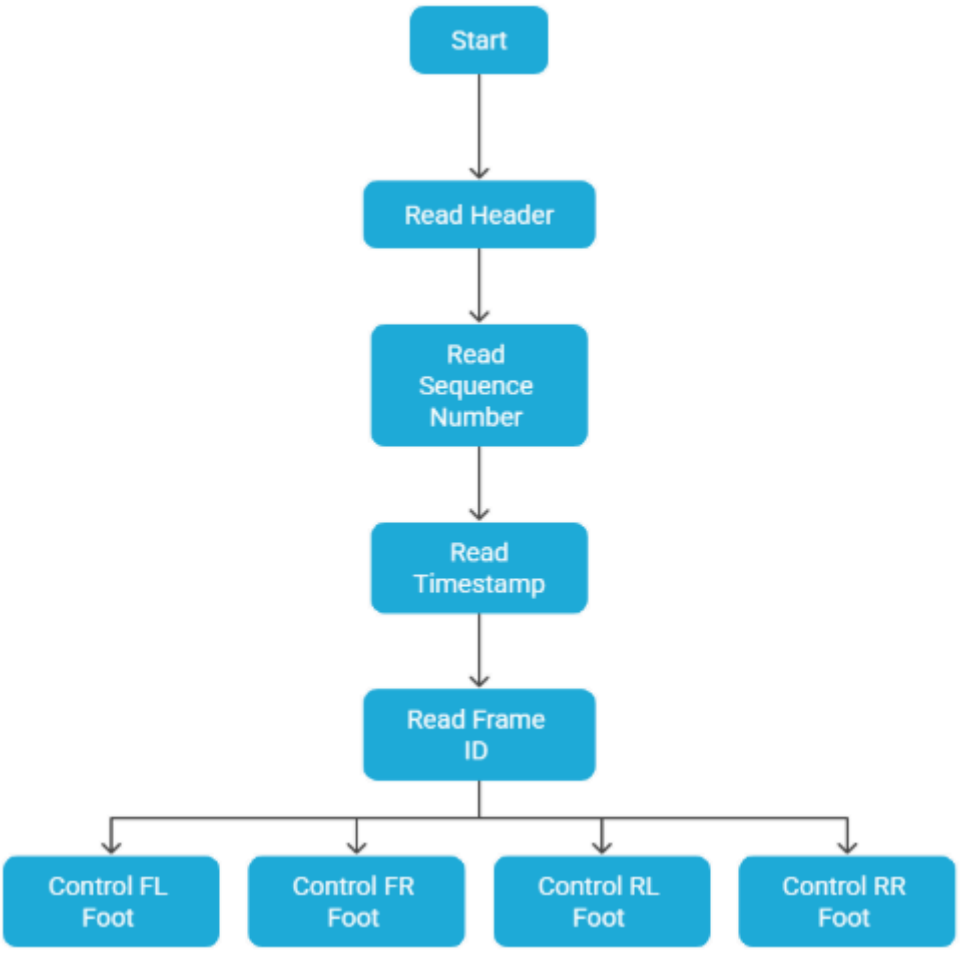
- Set `stamp` = current system time
- Set `frame_id` = `"base_link"` (or any robot frame)

Step 2: For each foot (FL, FR, RL, RR):

- Set desired angles for hip, knee, and ankle.
- These can be based on gait patterns (standing, walking, trotting).

Step 3: Package all into a message structure.

Flowchart:



3. `TaskSpace.msg`

How is it Created?

`TaskSpace.msg` defines the target 3D position (x, y, z) of each foot.

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id

geometry_msgs/Point FL_foot
geometry_msgs/Point FR_foot
geometry_msgs/Point RL_foot
geometry_msgs/Point RR_foot
```

Code Explanation

- **Header**: Same as in `JointSpace` for synchronization.
- **FL_foot, FR_foot, RL_foot, RR_foot**: Each is a `geometry_msgs/Point` representing a (x, y, z) location.

This message is **published by a higher-level controller** that plans in Cartesian space. The receiver must perform **Inverse Kinematics** to convert foot positions into joint angles.

Why Declared This Way

- Foot placement planning is more natural in **real-world coordinates** (meters) rather than joint angles.
- Enables complex motion planning like walking over uneven terrain.
- Dividing by leg allows **individual foot trajectories**.
- Keeping standard `Point` messages allows easy visualization in RViz and
- compatibility with many ROS tools.

Algorithm:

1. **Initialize** a new message object.
2. **Fill Header Information:**

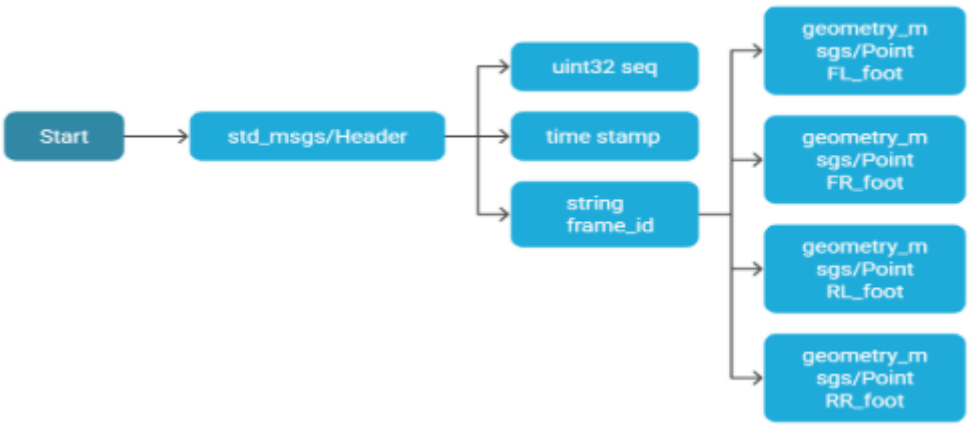
- Set `header.seq` to the current sequence number.
- Set `header.stamp` to the current ROS time.
- Set `header.frame_id` to the reference frame (e.g., `"base_link"` or `"odom"`).

3. **Assign Foot Positions:**

- Set `FL_foot.x = x1`, `FL_foot.y = y1`, `FL_foot.z = z1`.
- Set `FR_foot.x = x2`, `FR_foot.y = y2`, `FR_foot.z = z2`.
- Set `RL_foot.x = x3`, `RL_foot.y = y3`, `RL_foot.z = z3`.
- Set `RR_foot.x = x4`, `RR_foot.y = y4`, `RR_foot.z = z4`.

4. **Publish** the message to the appropriate ROS topic.
5. **Increment** the sequence number for the next message.
6. **End.**

Flowchart:



src/dingo folder:

The folder `dingo_ws/src/dingo_control/src/dingo_control` within the DingoQuadruped repository contains the core control logic for the Dingo quadruped robot. This directory is part of the `dingo_control` ROS package, which is responsible for implementing the robot's movement behaviors, including gait generation, kinematics, and trajectory planning.

Files inside src/dingo folder:

1. **command.py:**

Description: Defines the `Command` class to hold the robot's desired movement instructions.

```
import numpy as np

class Command:

    """Stores movement command"""

    def __init__(self):

        self.horizontal_velocity = np.array([0, 0])  # Desired x and y velocity.

        self.yaw_rate = 0.0                  # Desired turning speed.

        self.height = -0.28                  # Desired body height.

        self.pitch = 0.0                     # Desired forward/backward tilt.

        self.roll = 0.0                      # Desired sideways tilt.

        self.joystick_control_active = 0     # Flag for joystick control.

        self.trotting_active = 0             # Flag for trotting gait.

        self.height_movement = 0             # Relative height change.

        self.roll_movement = 0               # Relative roll change.

        self.hop_event = False               # Flag for hop action.

        self.trot_event = False              # Flag for trot action.

        self.joystick_control_event = False  # Flag for joystick control change.
```

Attributes (Concise Explanation)

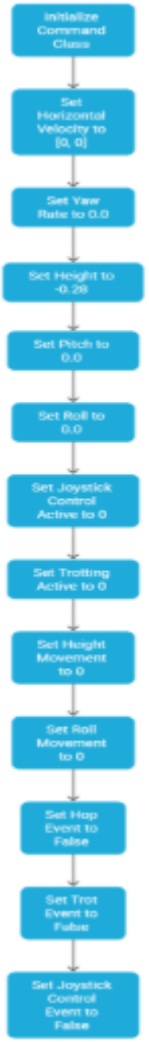
- **`horizontal_velocity`**: Desired ground speed and direction (x, y).
- **`yaw_rate`**: Desired turning rate.
- **`height`**: Desired body height.
- **`pitch`**: Desired forward/backward body tilt.
- **`roll`**: Desired sideways body tilt.
- **`joystick_control_active`**: Enables/disables joystick input.
- **`trotting_active`**: Enables/disables trotting gait.
- **`height_movement`**: Incremental height adjustment.
- **`roll_movement`**: Incremental roll adjustment.
- **`hop_event`**: Triggers a hop action.
- **`trot_event`**: Triggers a trot action/gait change.
- **`joystick_control_event`**: Indicates a change in joystick control status.

Algorithm:

1. Create a new `Command` object to store movement instructions.
2. Get user input from a joystick or keyboard:
    - Forward/backward speed
    - Sideways speed
    - Turn speed (rotate left/right)
    - Height change (up or down)
    - Body tilt (forward/backward or side-to-side)
    - Check if buttons like "hop" or "trot" are pressed
3. Update the Command object with these values:
    - Set `horizontal_velocity` to move forward/backward or sideways
    - Set `yaw_rate` to turn the robot

    - Adjust `height` to raise or lower the robot
    - Set `pitch` and `roll` for body tilting
    - Set flags if joystick is active or trotting is enabled
4. Check for special actions:
    - If the "hop" button is pressed, set `hop_event` to True
    - If the "trot" button is pressed, set `trot_event` to True
    - If the joystick control is turned on/off, set `joystick_control_event` to True
5. Send the Command object to the robot's controller so it knows what to do
6. (Optional) Reset special actions like `hop_event` to False after they are used
7. Repeat steps 3–7 in a loop (many times per second)

Flowchart:



2. Config.py

Description:

This file defines the `Configuration` class, which serves as a central repository for various parameters and settings essential for the Dingo robot's control system and simulation. These parameters govern the robot's movement capabilities, gait, physical dimensions, inertial properties, and simulation environment. Additionally, it defines the `Leg_linkage` class for storing leg-specific geometric parameters relevant for hardware interfacing.

```
01 | import numpy as np                        # Import NumPy library for numerical operations

02 | from dingo_input_interfacing.HardwareConfig import PS4_COLOR,
PS4_DEACTIVATED_COLOR  # Import color configs from controller setup

03 | from enum import Enum                     # Import Enum class for creating constant sets

04 | import math as m                          # Import math library as 'm' to access functions like
sin, cos, radians etc.


05 |

06 | class Configuration:                      # Class to store all robot parameters (movement,
gait, geometry, etc.)

07 |     def __init__(self):

08 |         self.ps4_color = PS4_COLOR          # Color of PS4 light when robot is active

09 |         self.ps4_deactivated_color = PS4_DEACTIVATED_COLOR  # Color when
controller is not in use


10 |         # COMMAND LIMITS

11 |         self.max_x_velocity = 1.2           # Max speed in X-direction (forward/backward)

12 |         self.max_y_velocity = 0.5           # Max speed in Y-direction (sideways)

13 |         self.max_yaw_rate = 2.0             # Max rotation speed (left/right turning)

14 |         self.max_pitch = 30.0 * np.pi / 180.0  # Max pitch (forward/backward tilt) in radians


15 |         # MOVEMENT PARAMETERS

16 |         self.z_time_constant = 0.02         # Time constant for vertical movement
smoothness

17 |         self.z_speed = 0.06                 # Max vertical speed (up/down)

18 |         self.pitch_deadband = 0.05          # Minimum pitch angle to start reacting

19 |         self.pitch_time_constant = 0.25     # Pitch reaction smoothness

20 |         self.max_pitch_rate = 0.3           # Max speed of pitch rotation

21 |         self.roll_speed = 0.1               # Max roll rotation speed (side-to-side)
```

```
22 |         self.yaw_time_constant = 0.3        # Yaw rotation smoothness
23 |         self.max_stance_yaw = 1.2           # Max rotation allowed during stance phase
24 |         self.max_stance_yaw_rate = 1        # Max rotation speed in stance

25 |         # STANCE (Default Foot Positions)
26 |         self.delta_x = 0.117                # Distance from body center to leg front-back
27 |         self.rear_leg_x_shift = -0.04       # Shift back legs slightly backward
28 |         self.front_leg_x_shift = 0.00       # Front legs remain in default
29 |         self.delta_y = 0.1106               # Distance from center to legs left-right
30 |         self.default_z_ref = -0.25          # Default ground height for foot (below body)

31 |         # SWING PHASE (Foot in air)
32 |         self.z_coeffs = None                # Placeholder for polynomial swing coefficients
33 |         self.z_clearance = 0.07             # Max height foot lifts off ground during swing
34 |         self.alpha = 0.5                    # Forward touch point ratio in swing
35 |         self.beta = 0.5                     # Backward touch point ratio in swing

36 |         # GAIT TIMING
37 |         self.dt = 0.01                      # Time step (10ms)
38 |         self.num_phases = 4                 # 4 phases in one gait cycle
39 |         self.contact_phases = np.array(     # Foot contact pattern in each phase
40 |             [[1, 1, 1, 0], [1, 0, 1, 1], [1, 0, 1, 1], [1, 1, 1, 0]]
41 |         )
42 |         self.overlap_time = 0.04            # Time when all 4 feet touch the ground
43 |         self.swing_time = 0.07              # Time when only 2 feet are on ground
```

```
44 |         # BODY GEOMETRY (Frame and leg origins)
45 |         self.LEG_FB = 0.11165              # Distance front-back from body center to leg
46 |         self.LEG_LR = 0.061                # Distance left-right from body center to leg
47 |         self.LEG_ORIGINS = np.array([       # Position of each leg relative to center
48 |             [self.LEG_FB, self.LEG_FB, -self.LEG_FB, -self.LEG_FB],
49 |             [-self.LEG_LR, self.LEG_LR, -self.LEG_LR, self.LEG_LR],
50 |             [0, 0, 0, 0],
51 |         ])

52 |         # LEG SEGMENT LENGTHS
53 |         self.L1 = 0.05162024721            # Hip segment length
54 |         self.L2 = 0.130                    # Upper leg length
55 |         self.L3 = 0.13813664159            # Lower leg length
56 |         self.phi = m.radians(73.91738698)    # Initial knee joint angle (from CAD)

57 |         # INERTIA SETTINGS
58 |         self.FRAME_MASS = 0.560            # Robot body mass
59 |         self.MODULE_MASS = 0.080           # Each servo module weight
60 |         self.LEG_MASS = 0.030              # Each leg's mass
61 |         self.MASS = self.FRAME_MASS + (self.MODULE_MASS + self.LEG_MASS) * 4  # Total robot mass

62 |         self.FRAME_INERTIA = tuple(map(lambda x: 3.0 * x, (1.844e-4, 1.254e-3, 1.337e-3)))  # Inertia of frame (scaled up)
63 |         self.MODULE_INERTIA = (3.698e-5, 7.127e-6, 4.075e-5)  # Inertia of servo module

64 |         # LEG INERTIA (simplified estimation)
```

```
65 |         leg_z = 1e-6                    # Very small z-axis inertia
66 |         leg_mass = 0.010                 # Mass used to compute inertia
67 |         leg_x = 1 / 12 * self.L2**2 * leg_mass  # Inertia of upper leg around x-axis
68 |         leg_y = leg_x                    # Same as x (symmetry)
69 |         self.LEG_INERTIA = (leg_x, leg_y, leg_z)

70 |
71 |     @property
72 |     def default_stance(self):            # Compute default leg positions relative to body
73 |         return np.array([
74 |             [self.delta_x + self.front_leg_x_shift, self.delta_x + self.front_leg_x_shift,
75 |              -self.delta_x + self.rear_leg_x_shift, -self.delta_x + self.rear_leg_x_shift],
76 |             [-self.delta_y, self.delta_y, -self.delta_y, self.delta_y],
77 |             [0, 0, 0, 0],
78 |         ])

79 |     @property
80 |     def z_clearance(self):               # Getter for foot swing height
81 |         return self.__z_clearance

82 |     @z_clearance.setter
83 |     def z_clearance(self, z):            # Setter for foot swing height
84 |         self.__z_clearance = z

85 |     @property
86 |     def overlap_ticks(self):             # Convert overlap time to ticks
```

```
87 |         return int(self.overlap_time / self.dt)

88 |     @property
89 |     def swing_ticks(self):                  # Convert swing time to ticks
90 |         return int(self.swing_time / self.dt)

91 |     @property
92 |     def stance_ticks(self):                 # Total time a foot is on the ground
93 |         return 2 * self.overlap_ticks + self.swing_ticks

94 |     @property
95 |     def phase_ticks(self):                  # Tick count for each gait phase
96 |         return np.array([self.overlap_ticks, self.swing_ticks, self.overlap_ticks,
self.swing_ticks])

97 |     @property
98 |     def phase_length(self):                 # Total ticks in one gait cycle
99 |         return 2 * self.overlap_ticks + 2 * self.swing_ticks
```

```
100| class SimulationConfig:                    # Separate simulation config (e.g., for MuJoCo)
101|     def __init__(self):
102|         self.XML_IN = "pupper.xml"          # Input robot model XML
103|         self.XML_OUT = "pupper_out.xml"     # Output robot model after sim
104|         self.START_HEIGHT = 0.3             # Initial height of robot in sim
105|         self.MU = 1.5                       # Coefficient of friction
```

```
106|        self.DT = 0.001                  # Time step of simulator
107|        self.JOINT_SOLREF = "0.001 1"        # Joint dynamics settings
108|        self.JOINT_SOLIMP = "0.9 0.95 0.001"  # Joint impulse solver
109|        self.GEOM_SOLREF = "0.01 1"          # Geometric contact solver ref
110|        self.GEOM_SOLIMP = "0.9 0.95 0.001"   # Geometric impulse solver

111|        # Servo & motor dynamics
112|        G = 220                       # Gear ratio
113|        m_rotor = 0.016                 # Rotor mass
114|        r_rotor = 0.005                 # Rotor radius
115|        self.ARMATURE = G**2 * m_rotor * r_rotor**2  # Moment of inertia

116|        NATURAL_DAMPING = 1.0
117|        ELECTRICAL_DAMPING = 0.049
118|        self.REV_DAMPING = NATURAL_DAMPING + ELECTRICAL_DA_
```

Algorithm:

1. Import Required Libraries

- Import `numpy` for array operations and math.
- Import `enum` for constant definitions.
- Import `math` for trigonometric and conversion operations.
- Import controller color settings from `HardwareConfig`.

2. Define `Configuration` Class

This class contains all robot parameters including control limits, body geometry, gait cycle, and inertial properties.

Initialize Parameters

1. Set controller light colors for active/inactive state.
2. Define max command limits: velocities, pitch angle.
3. Set movement tuning values: speeds, deadbands, time constants.
4. Specify stance configuration (foot position layout).
5. Set swing phase parameters like foot clearance and coefficients.
6. Define gait cycle:
    - Total time per phase.
    - Contact phase pattern of legs.
7. Configure robot frame and leg geometry:
    - Leg origin positions.
    - Segment lengths (hip, upper leg, lower leg).
8. Define inertia-related properties:
    - Masses of parts.
    - Moment of inertia of body and legs.

Define Property Methods

1. `default_stance` - Calculates the foot positions in default posture.
2. `z_clearance` - Getter/setter for swing height.
3. `overlap_ticks`, `swing_ticks`, `stance_ticks` - Convert times to ticks based on `dt`.
4. `phase_ticks` - Tick duration per gait phase.

5. `phase_length` - Total ticks in one complete gait cycle.

3. Define `SimulationConfig` Class

Contains MuJoCo or physics simulator-specific parameters:

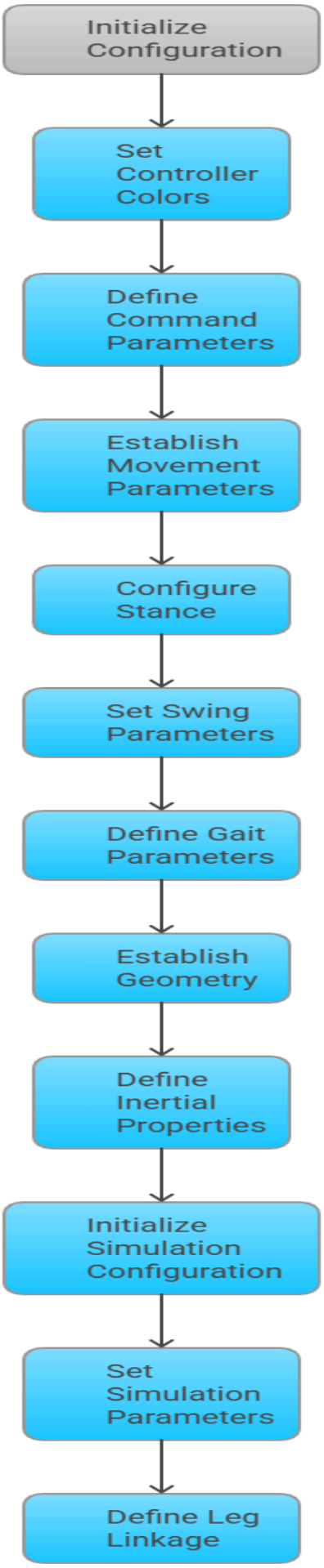
1. XML input/output files.
2. Initial position and physical constants like friction, time step.
3. Servo/motor modeling: damping, stiffness, torque limits.
4. Gear ratio-based armature inertia calculation.

4. Define `Leg_linkage` Class

Contains detailed leg linkage lengths from CAD:

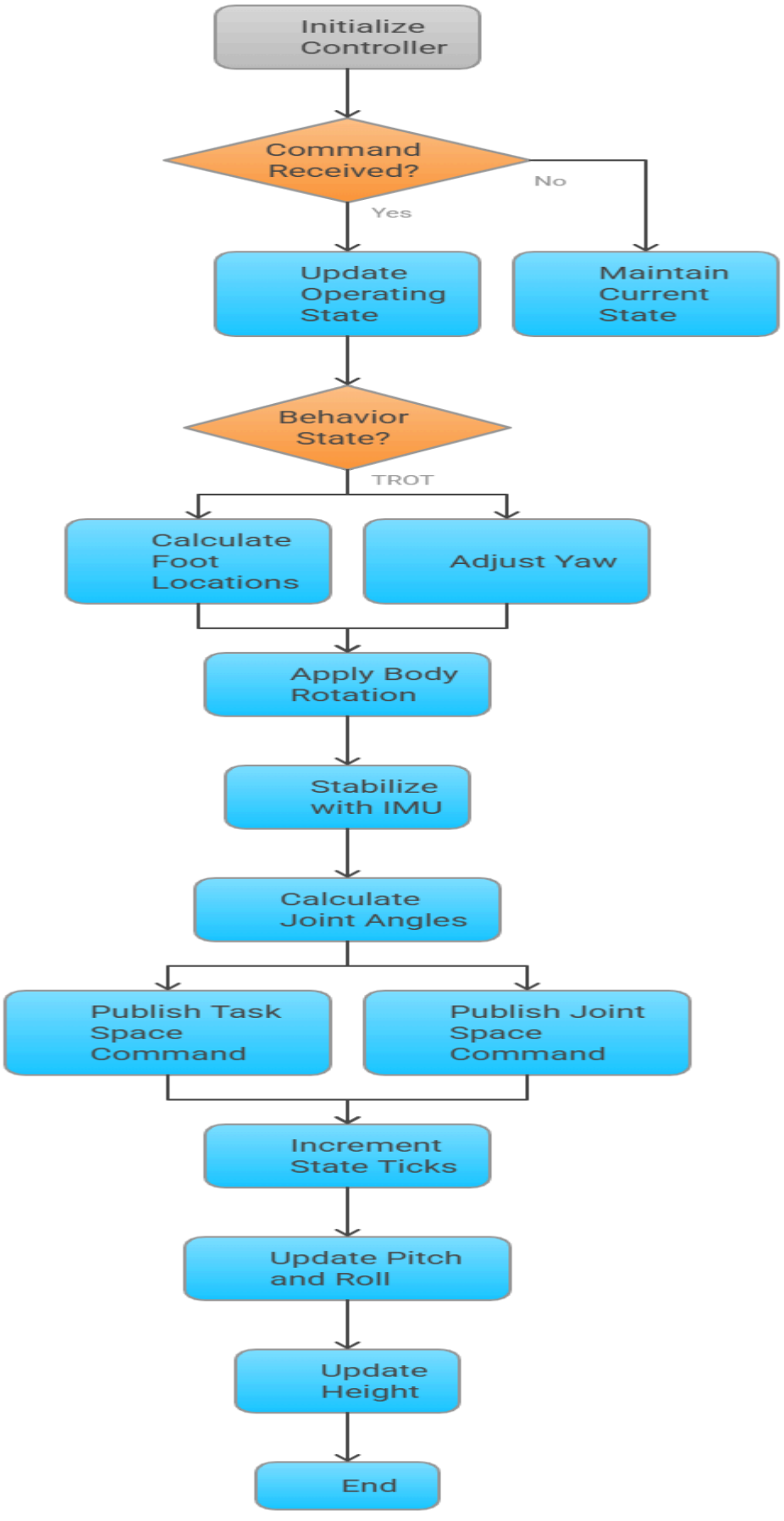
1. Load real-world measurements (`a` to `h`) in mm.
2. Fetch lengths from `Configuration` and convert meters to mm.
3. Calculate angles like `gamma` and triangle angle `EDC` using trigonometry.

Flowchart:


Initialize Configuration
Set Controller Colors
Define Command Parameters
Establish Movement Parameters
Configure Stance
Set Swing Parameters
Define Gait Parameters
Establish Geometry
Define Inertial Properties
Initialize Simulation Configuration
Set Simulation Parameters
Define Leg Linkage

Controllers.py:

Flowchart:



```
1  #!/usr/bin/env python3  # Tell the system to run the file using Python 3.

2  import rospy  # Import rospy to use ROS Python functionalities.

3  import numpy as np  # Import numpy for math operations like arrays and matrices.

4  from quadruped_control.controllers.swing_controller import SwingController  #
Import the Swing Controller class.

5  from quadruped_control.controllers.stance_controller import StanceController  #
Import the Stance Controller class.

6  from quadruped_control.controllers.gait_controller import GaitController  # Import
the Gait Controller class.

7  from quadruped_control.utilities.kinematics import four_legs_inverse_kinematics
# Import function to calculate joint angles from foot positions.

8  from quadruped_control.utilities.kinematics import euler2mat  # Import function to
create rotation matrices from Euler angles.

9  from quadruped_control.utilities.filters import clipped_first_order_filter  # Import
smoothing filter to avoid sudden changes.

10 from quadruped_control.msg import TaskSpace, JointSpace  # Import custom
ROS messages for task-space and joint-space control.

11 from geometry_msgs.msg import Point  # Import standard ROS message for
points (x, y, z).

12 from std_msgs.msg import Header  # Import standard ROS header message.

13 from quadruped_control.utilities.robot_config import BehaviorState  # Import
behavior states like TROT, HOP, etc.

14 class Controller(object):  # Define the Controller class to manage walking
behavior.

15     def __init__(self, config):  # Constructor method to initialize the Controller.

16         self.config = config  # Store the robot configuration.

17

18         self.swing_controller = SwingController(config)  # Initialize the Swing
Controller.
```

19 self.stance_controller = StanceController(config) # Initialize the Stance Controller.

20 self.gait_controller = GaitController(config) # Initialize the Gait Controller.

21

22 self.smoothed_yaw = 0.0 # Initialize smoothed yaw angle for smooth rotation.

23 self.task_space_pub = rospy.Publisher('task_space_goals', TaskSpace, queue_size=10) # Create ROS publisher for task-space commands.

24 self.joint_space_pub = rospy.Publisher('joint_space_goals', JointSpace, queue_size=10) # Create ROS publisher for joint-space commands.

25 def inverse_kinematics(self, goal_foot_positions): # Define method to compute joint angles from foot positions.

26 return four_legs_inverse_kinematics(goal_foot_positions, self.config) # Return the calculated joint angles.

27 def step_gait(self, state, command): # Define method to perform one step of gait planning.

28 self.gait_controller.step(command) # Update gait based on the command.

29

30 contact_modes = self.gait_controller.get_contact_modes() # Get which legs should be touching the ground.

31 self.contact_modes = contact_modes # Save the contact modes.

32

33 swing_times = self.gait_controller.get_swing_times() # Get swing durations for each leg.

34 swing_states = self.gait_controller.get_swing_states() # Get which legs are currently swinging.

```
35
36          swing_feet = self.swing_controller.get_swing_foot_trajectory(
37              state, command, swing_states, swing_times)  # Calculate swing leg foot trajectories.
38
39          stance_feet = self.stance_controller.get_stance_foot_location(
40              state, command, contact_modes)  # Calculate stance leg foot locations.
41
42          return stance_feet + swing_feet  # Return combined foot positions for both stance and swing legs.


43      def publish_task_space_command(self, goal_foot_positions):  # Define method to publish foot positions in task space.
44          task_space_command = TaskSpace()  # Create empty TaskSpace message.
45          task_space_command.header = Header(stamp=rospy.Time.now())  # Add timestamp to the message.
46
47          task_space_command.fl = Point(*goal_foot_positions[0, :])  # Set Front-Left leg position.
48          task_space_command.fr = Point(*goal_foot_positions[1, :])  # Set Front-Right leg position.
49          task_space_command.bl = Point(*goal_foot_positions[2, :])  # Set Back-Left leg position.
50          task_space_command.br = Point(*goal_foot_positions[3, :])  # Set Back-Right leg position.
51
52          self.task_space_pub.publish(task_space_command)  # Publish the task-space command to ROS.
```

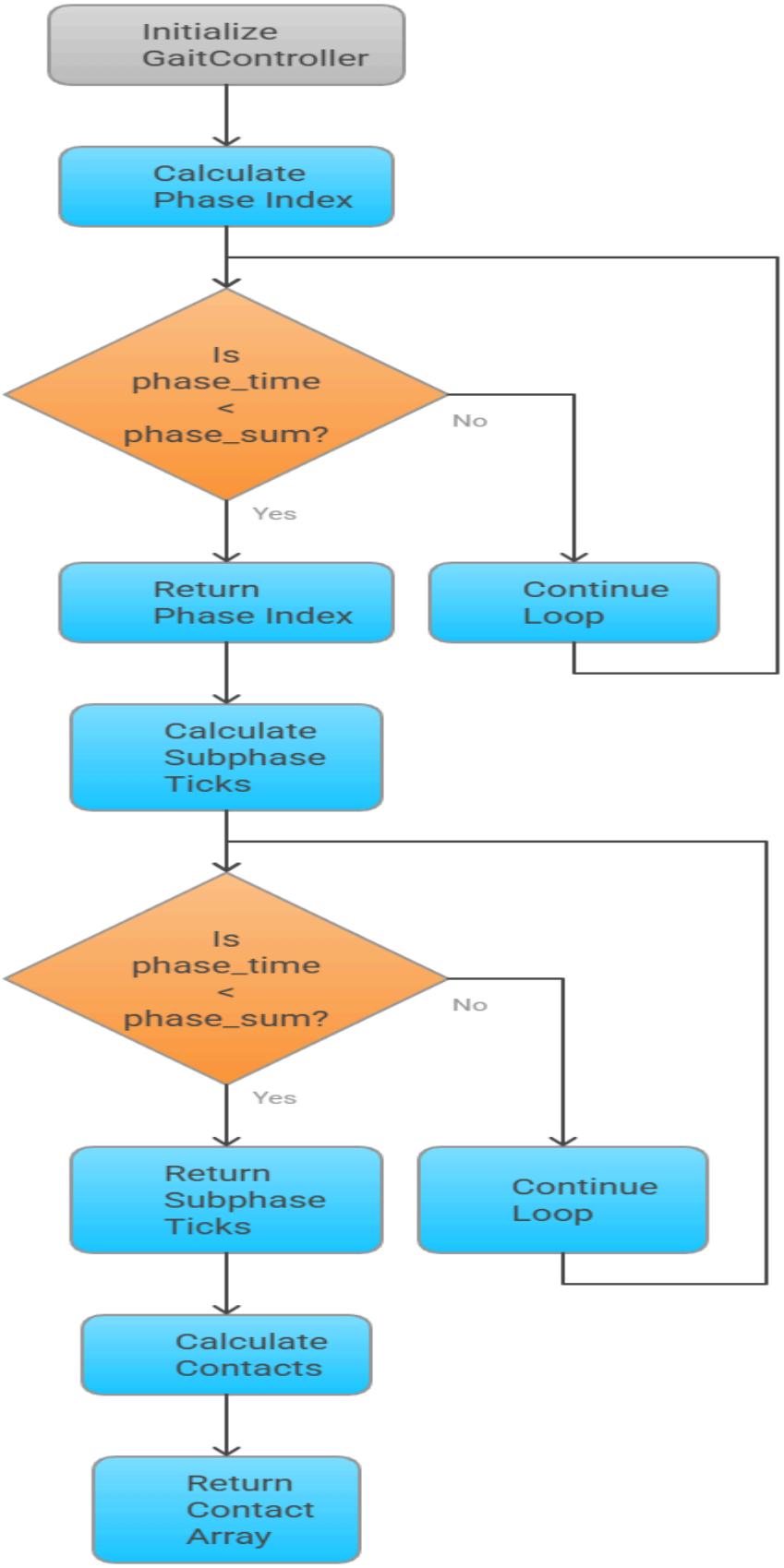
```
53     def publish_joint_space_command(self, joint_angles):  # Define method to publish joint angles in joint space.

54         joint_space_command = JointSpace()  # Create empty JointSpace message.

55         joint_space_command.header = Header(stamp=rospy.Time.now())  # Add timestamp to the message.

56

57         joint_space_command.fl = Angle(*np.rad2deg(joint_angles[0, :]))  # Set Front-Left joint angles (converted to degrees).

58         joint_space_command.fr = Angle(*np.rad2deg(joint_angles[1, :]))  # Set Front-Right joint angles (converted to degrees).

59         joint_space_command.bl = Angle(*np.rad2deg(joint_angles[2, :]))  # Set Back-Left joint angles (converted to degrees).

60         joint_space_command.br = Angle(*np.rad2deg(joint_angles[3, :]))  # Set Back-Right joint angles (converted to degrees).

61

62         self.joint_space_pub.publish(joint_space_command)  # Publish the joint-space command to ROS.



63     def run(self, state, command):  # Define the main method to control the robot every cycle.

64         if command.behavior_state == BehaviorState.TROT:  # If robot is in TROT mode,

65             goal_foot_positions = self.step_gait(state, command)  # Calculate foot positions using gait planning.

66

67         elif command.behavior_state == BehaviorState.HOP:  # If robot is in HOP mode,

68             goal_foot_positions = self.set_pose_to_default(state, height=0.18)  # Set legs to a lower fixed height.
```

```
69        elif command.behavior_state == BehaviorState.FINISHHOP:  # If robot is finishing hop,
70            goal_foot_positions = self.set_pose_to_default(state, height=0.23)  # Set legs to normal standing height.
71
72        else:  # If robot is resting or in unknown mode,
73            goal_foot_positions = self.stabilise_with_IMU(state, command)  # Stabilize robot using IMU data.


74        self.publish_task_space_command(goal_foot_positions)  # Publish target foot positions.
75
self.publish_joint_space_command(self.inverse_kinematics(goal_foot_positions))  # Calculate and publish joint angles.


76    def set_pose_to_default(self, state, height):  # Define method to set feet in a simple standing pose.
77        goal_foot_positions = np.zeros((4, 3))  # Create empty 4x3 array for four legs.
78        goal_foot_positions[:, 2] = -height  # Set height (z-axis) for all feet to -height.
79        return goal_foot_positions  # Return the default foot positions.


80    def stabilise_with_IMU(self, state, command):  # Define method to stabilize robot posture using IMU.
81        goal_foot_positions = np.zeros((4, 3))  # Create empty 4x3 array for foot positions.
82        goal_foot_positions[:, 2] = -0.23  # Set height for stable standing.
```

```
83        R = euler2mat(state.foot_locations_base_frame[0],
state.foot_locations_base_frame[1], 0.0)  # Create rotation matrix from roll and pitch
(ignore yaw).

84        for i in range(4):  # For each leg,

85            goal_foot_positions[i, :] = np.dot(R, state.default_stance[i, :])  # Rotate the
default stance to match body tilt.

86        self.smoothed_yaw = clipped_first_order_filter(

87            command.yaw_rate, self.smoothed_yaw, 0.05, self.config.dt)  # Smooth
the yaw rate input.

88        rotation_matrix = euler2mat(0, 0, self.smoothed_yaw * self.config.dt)  #
Create small rotation matrix for yaw.

89        for i in range(4):  # For each leg,

90            goal_foot_positions[i, :] = np.dot(rotation_matrix, goal_foot_positions[i, :])
# Apply small yaw rotation to each foot.

91        return goal_foot_positions  # Return stabilized foot positions.
```

Gait.py:

Flowchart:



```
1  class GaitController:  # Define the class that manages the gait cycle of the robot

2      def __init__(self, config):  # Constructor to initialize the GaitController with a config object

3          self.config = config  # Store the config in an instance variable to use throughout the class


4      def phase_index(self, ticks):  # Method to calculate which part of the gait cycle the robot should be in, based on time in ticks

5          """

6          Calculates which part of the gait cycle the robot should be in given the time in ticks.  # Documentation string explaining the purpose of the method

7

8          Parameters

9          ----------

10         ticks : int  # Number of timesteps since the program started

11         gaitparams : GaitParams  # GaitParams object (though not directly used here)

12

13         Returns

14         -------

15         Int  # The index of the gait phase the robot should be in

16         """


17         phase_time = ticks % self.config.phase_length  # Calculate the time within the current phase by finding the remainder of ticks divided by the phase length

18         phase_sum = 0  # Initialize a variable to keep track of the cumulative phase times

19
```

```
20        for i in range(self.config.num_phases):  # Loop through all the phases defined in the config

21            phase_sum += self.config.phase_ticks[i]  # Add the ticks of the current phase to the cumulative phase sum

22            if phase_time < phase_sum:  # If the current phase time is less than the cumulative phase sum, return the current phase index

23                return i

24        assert False  # If no phase is found, raise an assertion error


25    def subphase_ticks(self, ticks):  # Method to calculate the number of ticks since the start of the current phase

26        """

27        Calculates the number of ticks (timesteps) since the start of the current phase.  # Documentation string explaining the method's purpose

28

29        Parameters

30        ----------

31        ticks : Int  # Number of timesteps since the program started

32        gaitparams : GaitParams  # GaitParams object (not directly used here)

33

34        Returns

35        -------

36        Int  # The number of ticks since the start of the current phase

37        """


38        phase_time = ticks % self.config.phase_length  # Calculate the phase time by taking the remainder of ticks divided by phase length

39        phase_sum = 0  # Initialize phase_sum to track the cumulative phase times
```

```
40          subphase_ticks = 0  # Initialize subphase_ticks to store the number of ticks since the start of the phase
41
42          for i in range(self.config.num_phases):  # Loop through each phase to find the current phase and calculate subphase ticks
43              phase_sum += self.config.phase_ticks[i]  # Add the ticks for the current phase to the cumulative phase sum
44              if phase_time < phase_sum:  # If phase_time is less than phase_sum, calculate the subphase ticks
45                  subphase_ticks = phase_time - phase_sum + self.config.phase_ticks[i] # Calculate ticks since the start of the current phase
46                  return subphase_ticks  # Return the calculated subphase ticks
47          assert False  # If no subphase is found, raise an assertion error


48      def contacts(self, ticks):  # Method to calculate which feet should be in contact with the ground at the given ticks
49          """
50          Calculates which feet should be in contact at the given number of ticks.  # Documentation string for contacts method
51
52          Parameters
53          ----------
54          ticks : Int  # Number of timesteps since the program started
55          gaitparams : GaitParams  # GaitParams object (not directly used here)
56
57          Returns
58          -------
```

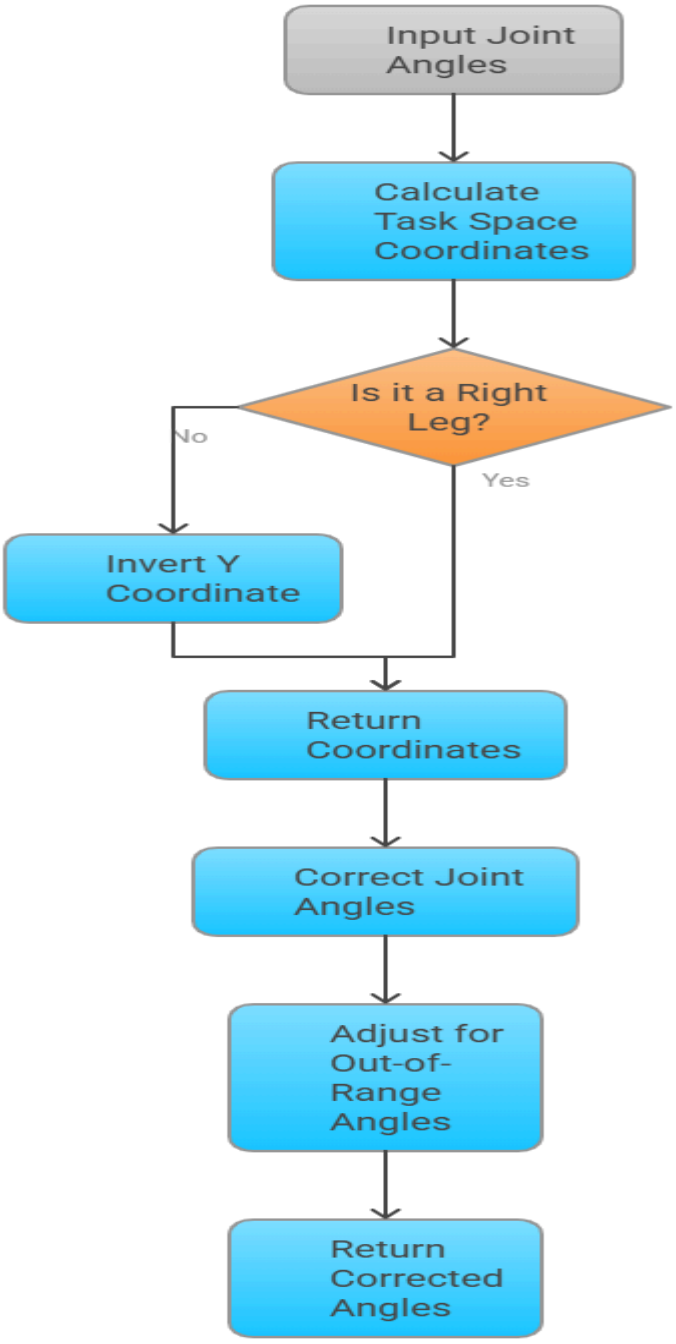
59 numpy array (4,) # Numpy array with 0 indicating flight and 1 indicating stance for each of the four feet

60 """

61 return self.config.contact_phases[:, self.phase_index(ticks)] # Return the contact phases for the current gait phase based on the phase index

Kinematics.py:

Flowchart:



```
1  import numpy as np  # Importing the numpy library for numerical operations

2  from numpy.linalg import inv, norm  # Importing functions for matrix inversion and norm calculations

3  from numpy import asarray, matrix  # Importing additional numpy functions

4  from math import *  # Importing all math functions (like pi, sin, cos, etc.)

5  #import matplotlib.pyplot as plt  # Commented out import for plotting (not used here)

6  from dingo_control.util import RotMatrix3D, point_to_rad  # Importing utilities from dingo_control for rotations and conversions

7  from transforms3d.euler import euler2mat  # Importing Euler angle to matrix conversion function

8  import rospy  # Importing rospy for ROS-related operations


9  def leg_explicit_inverse_kinematics(r_body_foot, leg_index, config):  # Function to calculate joint angles for a given foot position and leg configuration

10     """Find the joint angles corresponding to the given body-relative foot position for a given leg and configuration"""

11

12     if leg_index == 1 or leg_index == 3:  # Check if the leg is right (indices 1 or 3 are for right legs)

13         is_right = 0  # Set 'is_right' to 0 if it's a right leg

14     else:  # Otherwise, it's a left leg

15         is_right = 1  # Set 'is_right' to 1 if it's a left leg

16     x, y, z = r_body_foot[0], r_body_foot[1], r_body_foot[2]  # Extract the x, y, z coordinates from the input array

17     if is_right: y = -y  # If it's a right leg, flip the y-coordinate to adjust the calculations
```

```
18      r_body_foot = np.array([x, y, z])  # Create a numpy array from the adjusted coordinates

19      R1 = pi / 2 - config.phi  # Calculate the rotation angle based on the configuration (phi)

20      rot_mtx = RotMatrix3D([-R1, 0, 0], is_radians=True)  # Create a rotation matrix for rotating about the x-axis

21      r_body_foot_ = rot_mtx * (np.reshape(r_body_foot, [3, 1]))  # Apply the rotation to the foot position

22      r_body_foot_ = np.ravel(r_body_foot_)  # Flatten the rotated foot position

23      x = r_body_foot_[0]  # Extract the new x-coordinate after rotation

24      y = r_body_foot_[1]  # Extract the new y-coordinate after rotation

25      z = r_body_foot_[2]  # Extract the new z-coordinate after rotation

26      len_A = norm([0, y, z])  # Calculate the length of the projection of the vector onto the YZ plane

27      a_1 = point_to_rad(y, z)  # Calculate the angle between the positive y-axis and the end-effector in the YZ plane

28      a_2 = asin(sin(config.phi) * config.L1 / len_A)  # Calculate the angle between the leg's projection line and the YZ plane

29      a_3 = pi - a_2 - config.phi  # Calculate the third angle based on the geometry of the leg

30      if is_right: theta_1 = a_1 + a_3  # If it's a right leg, calculate theta_1

31      else: theta_1 = a_1 + a_3  # If it's a left leg, the formula for theta_1 is the same

32      if theta_1 >= 2 * pi: theta_1 = np.mod(theta_1, 2 * pi)  # Ensure theta_1 is within the range [0, 2pi]

33      offset = np.array([0.0, config.L1 * cos(theta_1), config.L1 * sin(theta_1)])  # Calculate the offset due to link 1

34      translated_frame = r_body_foot_ - offset  # Apply the offset to the rotated foot position
```

```
35      if is_right: R2 = theta_1 + config.phi - pi / 2  # Calculate the rotation angle for the second rotation (right leg)

36      else: R2 = -(pi / 2 - config.phi + theta_1)  # Calculate the second rotation angle for left leg (may need adjustment)

37      R2 = theta_1 + config.phi - pi / 2  # Update R2 calculation

38      rot_mtx = RotMatrix3D([-R2, 0, 0], is_radians=True)  # Create a new rotation matrix for rotating about the x-axis

39      j4_2_vec_ = rot_mtx * (np.reshape(translated_frame, [3, 1]))  # Apply the second rotation to the translated frame

40      j4_2_vec_ = np.ravel(j4_2_vec_)  # Flatten the rotated vector

41      x_, y_, z_ = j4_2_vec_[0], j4_2_vec_[1], j4_2_vec_[2]  # Extract the new x, y, z coordinates after the second rotation

42      len_B = norm([x_, 0, z_])  # Calculate the length of the projection of the vector onto the XZ plane

43      if len_B >= (config.L2 + config.L3):  # Check if the point is too far to reach

44          len_B = (config.L2 + config.L3) * 0.8  # If too far, limit the distance to 80% of the maximum reach

45          rospy.logwarn('target coordinate: [%f %f %f] too far away', x, y, z)  # Log a warning message

46      b_1 = point_to_rad(x_, z_)  # Calculate the angle between the positive x-axis and len_B in the XZ plane

47      b_2 = acos((config.L2**2 + len_B**2 - config.L3**2) / (2 * config.L2 * len_B))  # Calculate the angle between len_B and link 2

48      b_3 = acos((config.L2**2 + config.L3**2 - len_B**2) / (2 * config.L2 * config.L3))  # Calculate the angle between link 2 and link 3

49      theta_2 = b_1 - b_2  # Calculate the second joint angle (theta_2)

50      theta_3 = pi - b_3  # Calculate the third joint angle (theta_3)

51      angles = angle_corrector(angles=[theta_1, theta_2, theta_3])  # Apply angle corrections to the calculated joint angles

52      return np.array(angles)  # Return the final calculated joint angles
```

```
53  def four_legs_inverse_kinematics(r_body_foot, config):  # Function to calculate joint angles for all four legs
54      """Find the joint angles for all twelve DOF corresponding to the given matrix of body-relative foot positions."""
55
56      alpha = np.zeros((3, 4))  # Initialize a matrix to store the joint angles for each leg
57      for i in range(4):  # Loop through all four legs
58          body_offset = config.LEG_ORIGINS[:, i]  # Get the body offset for the current leg
59          alpha[:, i] = leg_explicit_inverse_kinematics(  # Calculate the joint angles for the current leg
60              r_body_foot[:, i] - body_offset, i, config  # Adjust the foot position by subtracting the body offset
61          )
62      return alpha  # Return the matrix of joint angles for all four legs
63  def forward_kinematics(angles, config, is_right=0):  # Function to calculate the foot position based on joint angles
64      """Find the foot position corresponding to the given joint angles for a given leg and configuration."""
65
66      x = config.L3 * sin(angles[1] + angles[2]) - config.L2 * cos(angles[1])  # Calculate the x-coordinate of the foot position
67      y = 0.5 * config.L2 * cos(angles[0] + angles[1]) - config.L1 * cos(angles[0] + (403 * pi) / 4500) - 0.5 * config.L2 * cos(angles[0] - angles[1]) - config.L3 * cos(angles[1] + angles[2]) * sin(angles[0])  # Calculate the y-coordinate
68      z = 0.5 * config.L2 * sin(angles[0] - angles[1]) + config.L1 * sin(angles[0] + (403 * pi) / 4500) - 0.5 * config.L2 * sin(angles[0] + angles[1]) - config.L3 * cos(angles[1] + angles[2]) * cos(angles[0])  # Calculate the z-coordinate
69      if not is_right:  # If it's not a right leg, flip the y-coordinate
```

```
70          y = -y

71      return np.array([x, y, z])  # Return the calculated foot position


72  def angle_corrector(angles=[0, 0, 0]):  # Function to correct joint angles to match the robot's configuration

73      angles[0] = angles[0]  # No change to the first angle (theta_1)

74      angles[1] = angles[1] - pi  # Adjust the second angle (theta_2) by subtracting pi

75      angles[2] = angles[2] - pi / 2  # Adjust the third angle (theta_3) by subtracting pi/2


76      for index, theta in enumerate(angles):  # Loop through each angle to ensure it's within the valid range

77          if theta > 2 * pi: angles[index] = np.mod(theta, 2 * pi)  # Ensure angle is within [0, 2pi]

78          if theta > pi: angles[index] = -(2 * pi - theta)  # Adjust angles greater than pi to be between -pi and pi

79      return angles  # Return the corrected angles
```

StanceController.py:

1. import numpy as np # Importing the numpy library for numerical computations, especially matrix operations.

2. from transforms3d.euler import euler2mat # Importing the euler2mat function to convert Euler angles to rotation matrices.

3. class StanceController: # Defining the StanceController class that will control the stance of the robot's legs.

4. def __init__(self, config): # Constructor method for initializing the class with a configuration.

5. self.config = config # Assigning the configuration object to the instance variable `config`.

6. def position_delta(self, leg_index, state, command): # Method to calculate the position and rotation change.

7. """Calculate the difference between the next desired body location and the current body location"""

8. z = state.foot_locations[2, leg_index] # Get the Z-coordinate (height) of the foot for the given leg.

9. v_xy = np.array(# Create a numpy array for the velocity in the XY plane.

10. [# The velocity vector contains three components:

11. -command.horizontal_velocity[0], # Negative horizontal velocity in the X direction.

12. -command.horizontal_velocity[1], # Negative horizontal velocity in the Y direction.

13. 1.0 / self.config.z_time_constant * (state.height - z), # The vertical velocity component (height difference).

14.]

15.)

16. delta_p = v_xy * self.config.dt # Calculate the position increment by multiplying velocity by the time step.

17. delta_R = euler2mat(0, 0, -command.yaw_rate * self.config.dt) # Calculate the rotation increment using Euler angles.

18. return (delta_p, delta_R) # Return the position and rotation increments as a tuple.

19. def next_foot_location(self, leg_index, state, command): # Method to calculate the next foot location.

20. foot_location = state.foot_locations[:, leg_index] # Get the current foot location for the given leg.

21. (delta_p, delta_R) = self.position_delta(leg_index, state, command) # Get the position and rotation increments.

22. incremented_location = delta_R @ foot_location + delta_p # Apply the position and rotation increments to get the new foot location.

23. return incremented_location # Return the incremented foot location.

Algorithm:

1. Initialize Controller

- Take in configuration settings when creating the controller.

2. Compute Position Delta

- Input: Leg index, robot's current state, and user command.
- Read the **current vertical (Z) position** of the selected foot.
- Compute how much the foot should move in X, Y, and Z directions:
    - X and Y are based on horizontal velocity commands.

    - Z is based on the difference between desired height and current foot height.

- Scale this velocity using the controller's time step to get the **position change** (`delta_p`).

- Compute the **rotation change** (`delta_R`) due to turning (yaw rate).

3. Compute Next Foot Location

- Input: Leg index, robot state, and command.

- Get the **current foot position** for the selected leg.

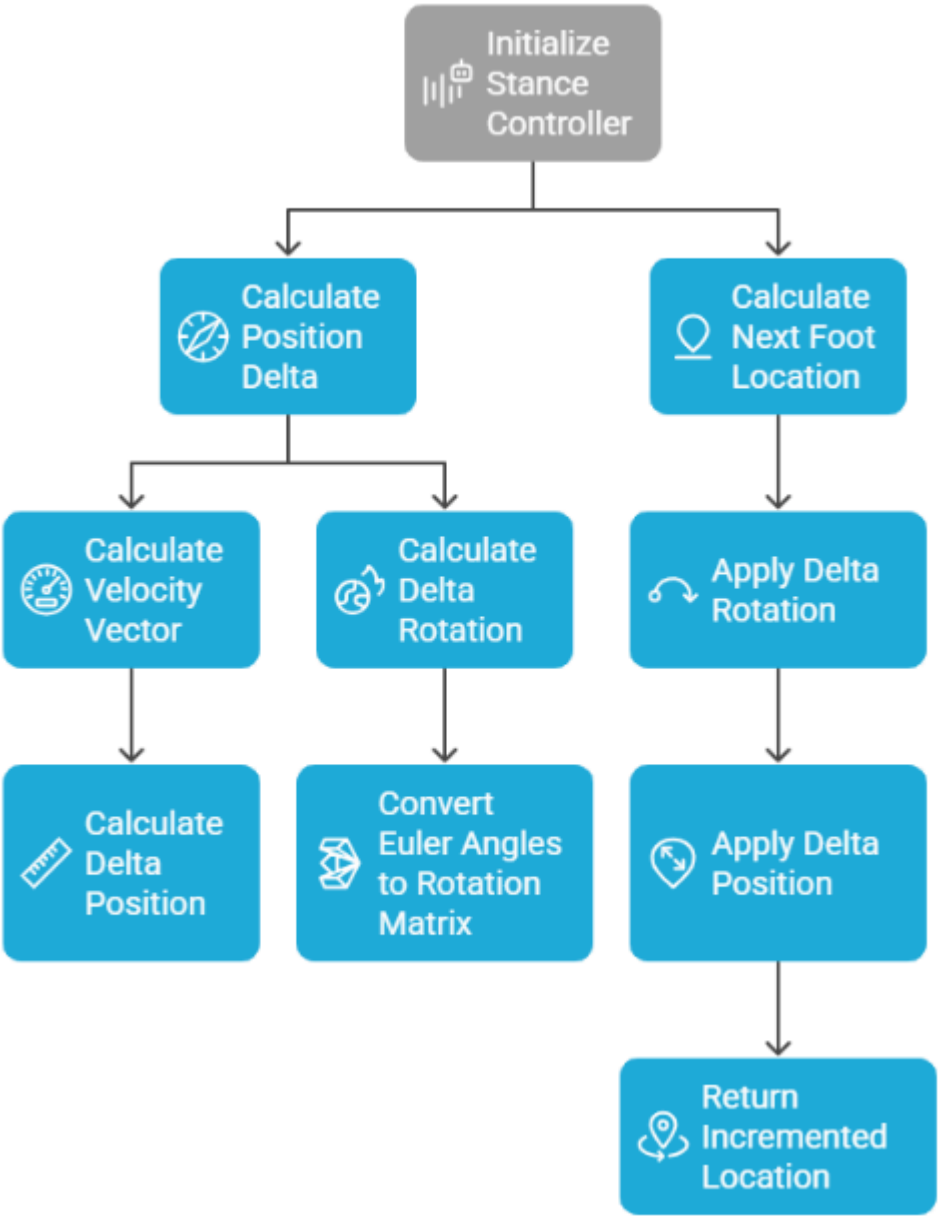
- Apply the rotation and translation to this position:

    - Multiply the current position with the rotation matrix.

    - Add the position change vector.

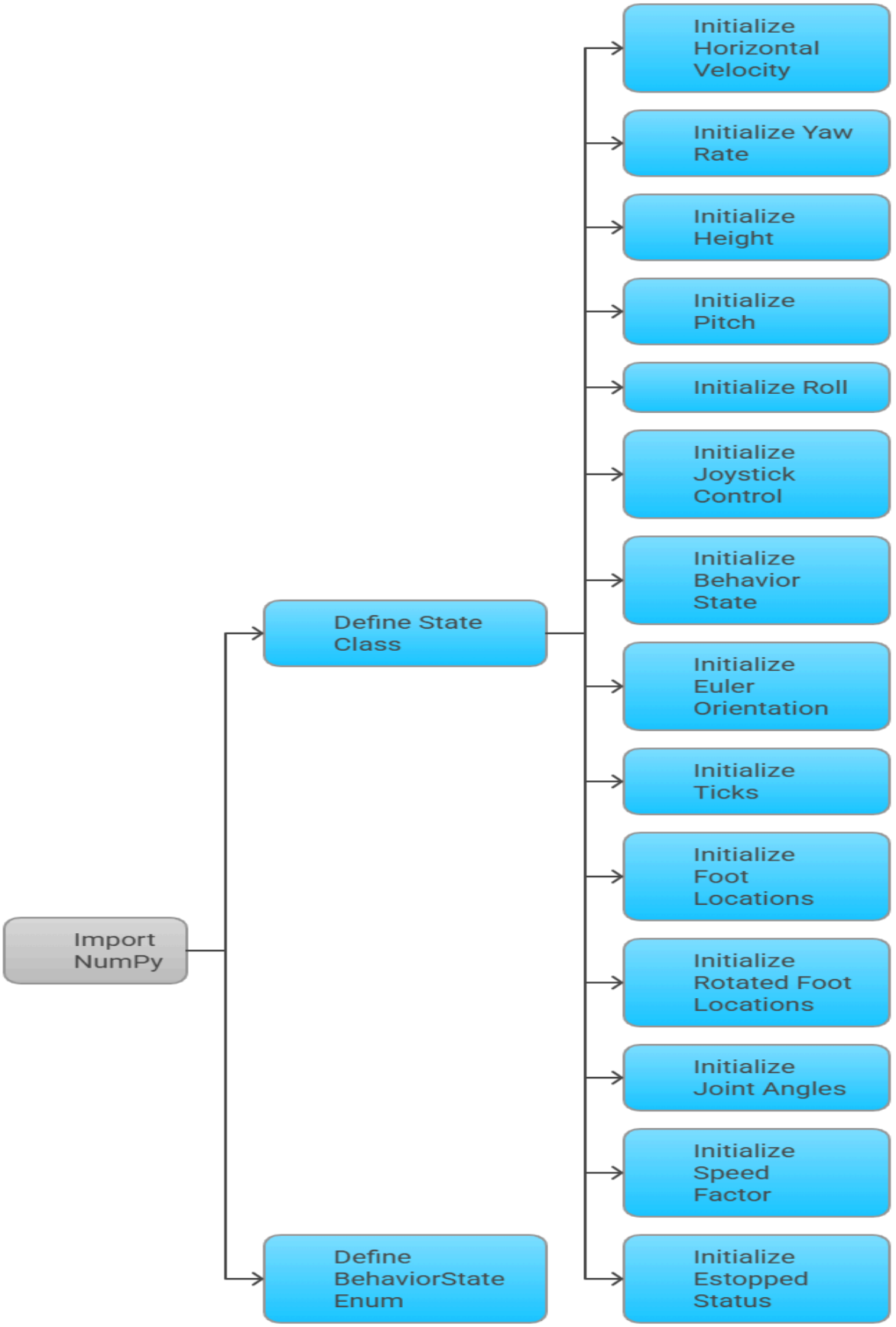
- Return the **new foot position**.

Flowchart:


Initialize Stance Controller
Calculate Position Delta
Calculate Next Foot Location
Calculate Velocity Vector
Calculate Delta Rotation
Apply Delta Rotation
Calculate Delta Position
Convert Euler Angles to Rotation Matrix
Apply Delta Position
Return Incremented Location

State.py:

Flowchart:



1. import numpy as np # Importing the numpy library for numerical computations.

2. from enum import Enum # Importing Enum to define a set of named constants for behavior states.

3. class State: # Defining the State class, which holds the current state of the robot.

4. def __init__(self): # Constructor method to initialize the State class with default values.

5. self.horizontal_velocity = np.array([0.0, 0.0]) # Initializing horizontal velocity of the robot (X and Y axes).

6. self.yaw_rate = 0.0 # Initializing the yaw rate (rotation around the Z axis) to 0.

7. self.height = -0.20 # Initializing the height of the robot's body from the ground.

8. self.pitch = 0.0 # Initializing the pitch angle (rotation around the X axis) to 0.

9. self.roll = 0.0 # Initializing the roll angle (rotation around the Y axis) to 0.

10. self.joystick_control_active = 1 # Indicates whether joystick control is active (1 = active, 0 = inactive).

11. self.behavior_state = BehaviorState.REST # Initializing the behavior state to REST (robot is idle).

12. self.euler_orientation = [0, 0, 0] # Initializing the Euler angles [pitch, roll, yaw] to 0.

13. self.ticks = 0 # Initializing the tick counter (for time steps or cycles).

14. self.foot_locations = np.zeros((3, 4)) # Initializing foot locations for 4 legs in 3D space (x, y, z).

15. self.rotated_foot_locations = np.zeros((3, 4)) # Initializing rotated foot locations for 4 legs.

16. self.joint_angles = np.zeros((3, 4)) # Initializing joint angles for 4 legs (theta_1, theta_2, theta_3).

17. self.speed_factor = 1 # Initializing the speed factor to 1 (no scaling).

18. self.currently_estopped = 0 # Flag to indicate if the robot is in an emergency stop state (0 = no, 1 = yes).

19. class BehaviorState(Enum): # Defining an enumeration for different behavior states of the robot.

20. DEACTIVATED = -1 # Behavior state representing a deactivated or turned-off robot.

21. REST = 0 # Behavior state representing the robot being at rest or idle.

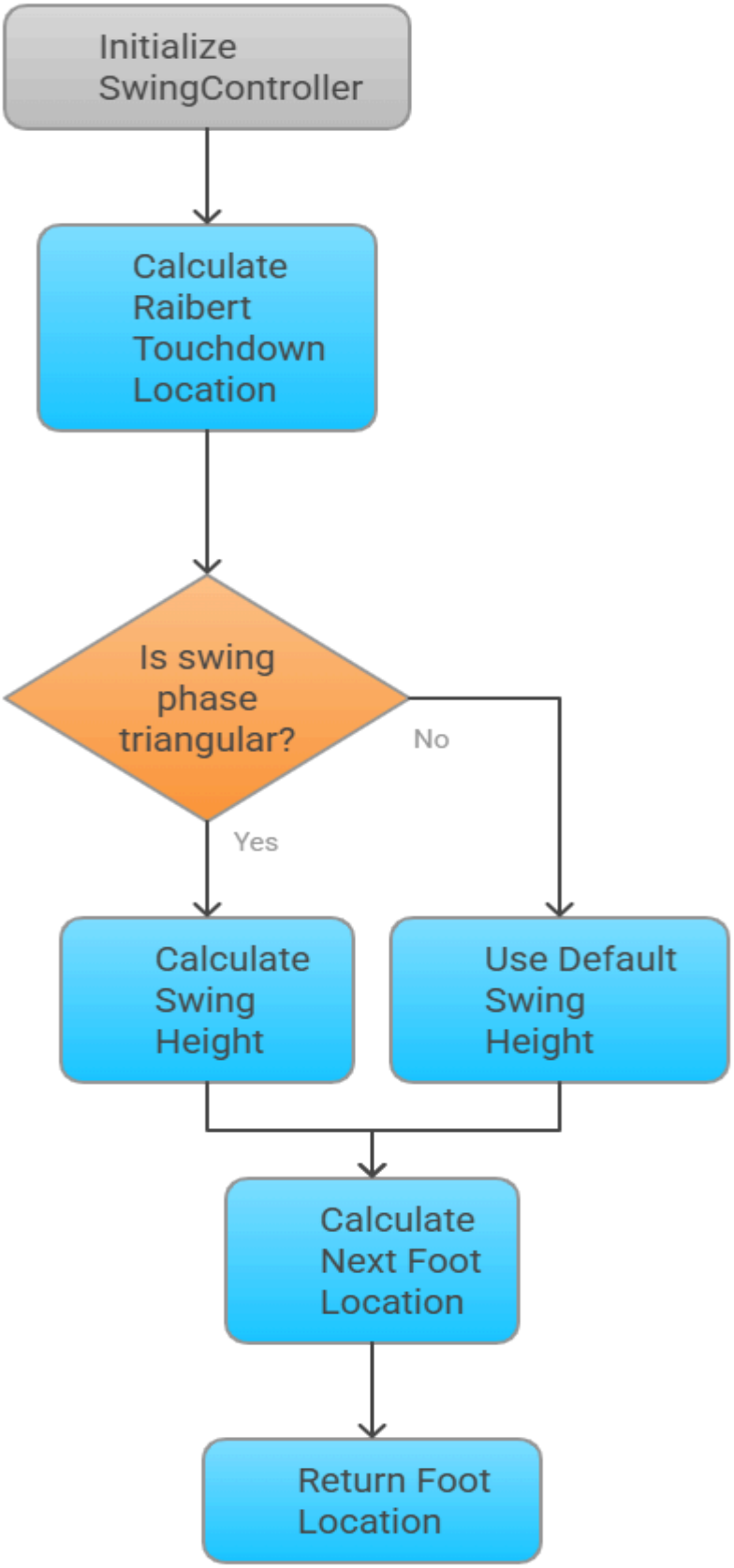
22. TROT = 1 # Behavior state representing the robot moving in a trot gait.

23. HOP = 2 # Behavior state representing the robot performing a hopping motion.

24. FINISHHOP = 3 # Behavior state representing the robot finishing a hop.

SwingLegController.py:

Flowchart:



1. import numpy as np # Importing numpy for numerical operations.

2. from transforms3d.euler import euler2mat # Importing euler2mat function to convert Euler angles to rotation matrices.

3. class SwingController: # Defining the SwingController class that controls the leg swing movement.

4. def __init__(self, config): # Constructor method initializing the SwingController with a configuration object.

5. self.config = config # Storing the configuration object to the instance variable for later use.

6. def raibert_touchdown_location(self, leg_index, command): # Method to compute the touchdown location based on the Raibert principle.

7. delta_p_2d = (# Calculating the 2D position change in the horizontal plane (x, y).

8. self.config.alpha # Scaling factor for horizontal velocity in the x and y directions.

9. * self.config.stance_ticks # Number of ticks in the stance phase.

10. * self.config.dt # Time step for each tick.

11. * command.horizontal_velocity # The commanded horizontal velocity for the robot.

12.)

13. delta_p = np.array([delta_p_2d[0], delta_p_2d[1], 0]) # Creating a 3D vector for the position change with no change in the z-direction.

14. theta = (# Calculating the rotation angle for yaw (rotation around the z-axis).

15. self.config.beta # Scaling factor for the yaw rate.

16. * self.config.stance_ticks # Number of stance ticks.

17. * self.config.dt # Time step for each tick.

18. * command.yaw_rate # The commanded yaw rate for the robot.

19.)

20. R = euler2mat(0, 0, theta) # Creating a rotation matrix using Euler angles (yaw rotation).

21. return R @ self.config.default_stance[:, leg_index] + delta_p # Returning the final touchdown location after applying rotation and translation.

22. def swing_height(self, swing_phase, triangular=True): # Method to calculate the swing height during the swing phase.

23. if triangular: # If the swing follows a triangular pattern.

24. if swing_phase < 0.5: # If the swing phase is less than 50%, the height increases.

25. swing_height_ = swing_phase / 0.5 * self.config.z_clearance # Increase swing height linearly.

26. else: # If the swing phase is greater than 50%, the height decreases.

27. swing_height_ = self.config.z_clearance * (1 - (swing_phase - 0.5) / 0.5) # Decrease swing height linearly.

28. return swing_height_ # Returning the calculated swing height.

29. def next_foot_location(self, swing_prop, leg_index, state, command): # Method to calculate the next foot location during the swing phase.

30. assert swing_prop >= 0 and swing_prop <= 1 # Ensuring the swing proportion is between 0 and 1.

31. foot_location = state.foot_locations[:, leg_index] # Getting the current foot location from the state object.

32. swing_height_ = self.swing_height(swing_prop) # Calculating the swing height for the given swing proportion.

33. touchdown_location = self.raibert_touchdown_location(leg_index, command) # Getting the touchdown location based on Raibert's model.

34. time_left = self.config.dt * self.config.swing_ticks * (1.0 - swing_prop) # Calculating the time remaining for the swing phase.

35. v = (touchdown_location - foot_location) / time_left * np.array([1, 1, 0]) # Calculating the velocity vector towards the touchdown location.

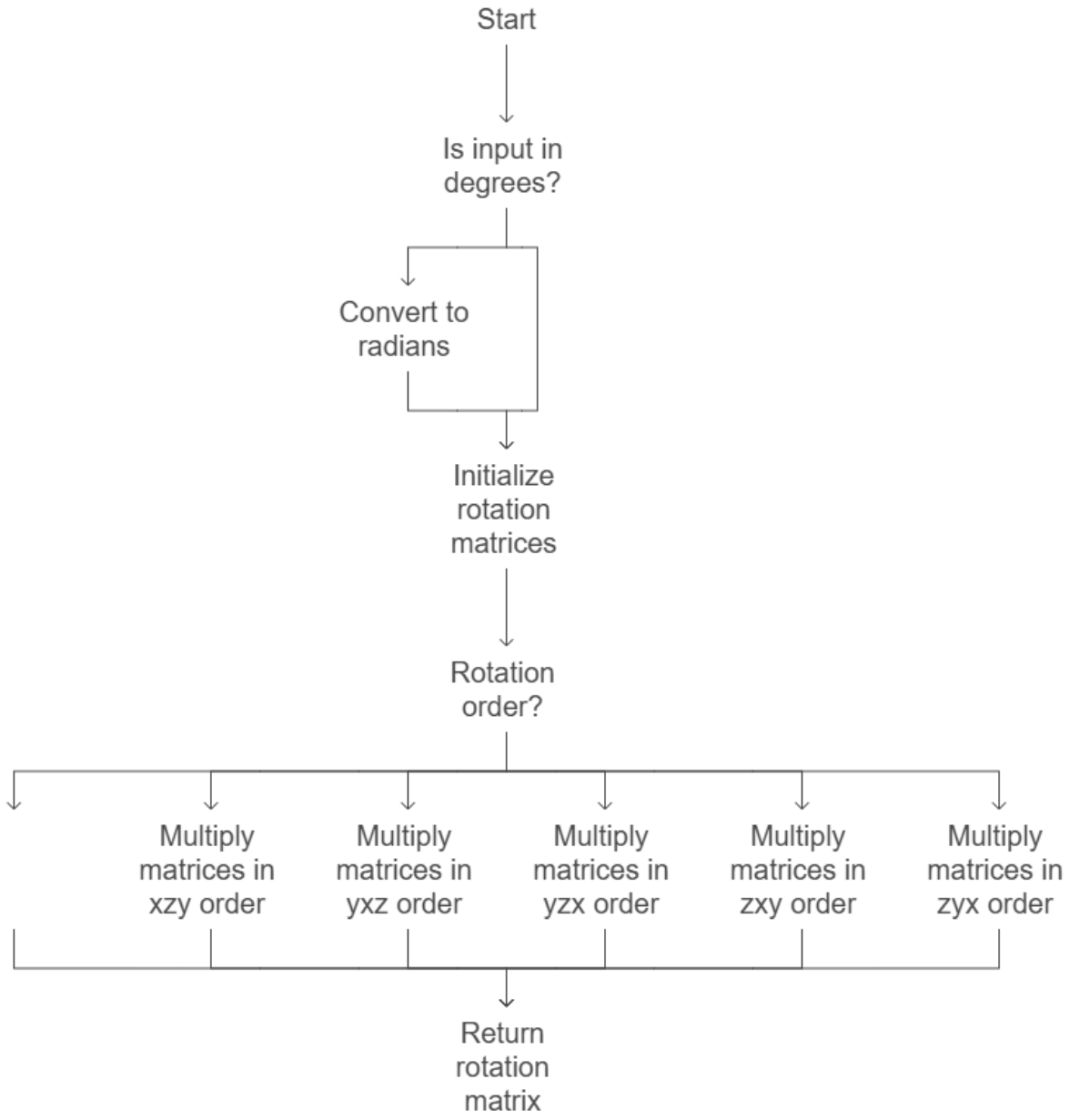
36. delta_foot_location = v * self.config.dt # Calculating the position increment for the foot during the swing phase.

37. z_vector = np.array([0, 0, swing_height_ + command.height]) # Creating a vector for the z-axis with swing height and commanded height.

38. return foot_location * np.array([1, 1, 0]) + z_vector + delta_foot_location # Returning the new foot location after applying position and height changes.

util.py:

Flowchart:



1. import numpy as np # Importing numpy for numerical operations.

2. from transforms3d.euler import euler2mat # Importing euler2mat function to convert Euler angles to rotation matrices.

3. class SwingController: # Defining the SwingController class that controls the leg swing movement.

4. def __init__(self, config): # Constructor method initializing the SwingController with a configuration object.

5. self.config = config # Storing the configuration object to the instance variable for later use.

6. def raibert_touchdown_location(self, leg_index, command): # Method to compute the touchdown location based on the Raibert principle.

7. delta_p_2d = (# Calculating the 2D position change in the horizontal plane (x, y).

8. self.config.alpha # Scaling factor for horizontal velocity in the x and y directions.

9. * self.config.stance_ticks # Number of ticks in the stance phase.

10. * self.config.dt # Time step for each tick.

11. * command.horizontal_velocity # The commanded horizontal velocity for the robot.

12.)

13. delta_p = np.array([delta_p_2d[0], delta_p_2d[1], 0]) # Creating a 3D vector for the position change with no change in the z-direction.

14. theta = (# Calculating the rotation angle for yaw (rotation around the z-axis).

15. self.config.beta # Scaling factor for the yaw rate.

16. * self.config.stance_ticks # Number of stance ticks.

17. * self.config.dt # Time step for each tick.

```
18.             * command.yaw_rate  # The commanded yaw rate for the robot.

19.         )

20.         R = euler2mat(0, 0, theta)  # Creating a rotation matrix using Euler angles (yaw rotation).

21.         return R @ self.config.default_stance[:, leg_index] + delta_p  # Returning the final touchdown location after applying rotation and translation.


22.     def swing_height(self, swing_phase, triangular=True):  # Method to calculate the swing height during the swing phase.

23.         if triangular:  # If the swing follows a triangular pattern.

24.             if swing_phase < 0.5:  # If the swing phase is less than 50%, the height increases.

25.                 swing_height_ = swing_phase / 0.5 * self.config.z_clearance  # Increase swing height linearly.

26.             else:  # If the swing phase is greater than 50%, the height decreases.

27.                 swing_height_ = self.config.z_clearance * (1 - (swing_phase - 0.5) / 0.5)  # Decrease swing height linearly.

28.         return swing_height_  # Returning the calculated swing height.


29.     def next_foot_location(self, swing_prop, leg_index, state, command):  # Method to calculate the next foot location during the swing phase.

30.         assert swing_prop >= 0 and swing_prop <= 1  # Ensuring the swing proportion is between 0 and 1.

31.         foot_location = state.foot_locations[:, leg_index]  # Getting the current foot location from the state object.

32.         swing_height_ = self.swing_height(swing_prop)  # Calculating the swing height for the given swing proportion.

33.         touchdown_location = self.raibert_touchdown_location(leg_index, command)  # Getting the touchdown location based on Raibert's model.
```

34. time_left = self.config.dt * self.config.swing_ticks * (1.0 - swing_prop) # Calculating the time remaining for the swing phase.

35. v = (touchdown_location - foot_location) / time_left * np.array([1, 1, 0]) # Calculating the velocity vector towards the touchdown location.

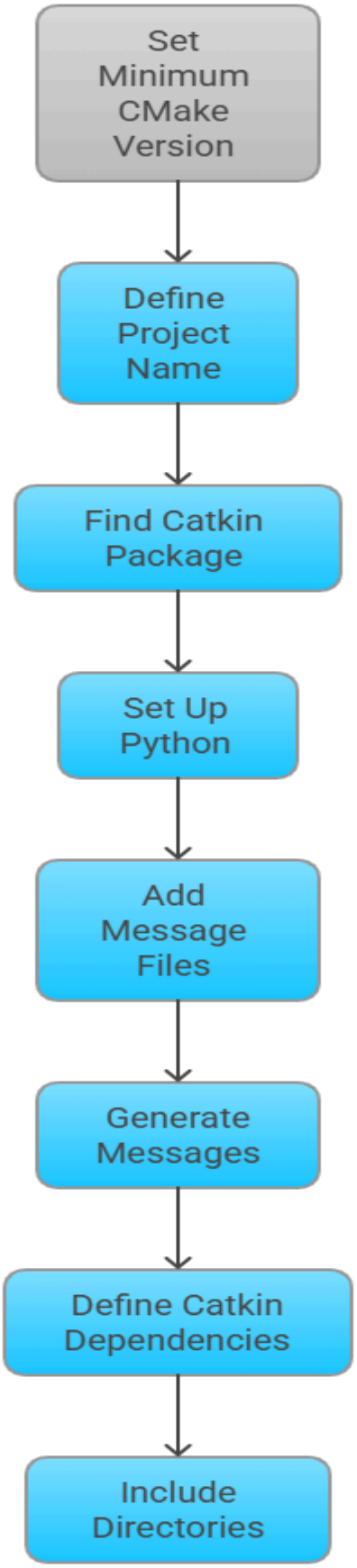
36. delta_foot_location = v * self.config.dt # Calculating the position increment for the foot during the swing phase.

37. z_vector = np.array([0, 0, swing_height_ + command.height]) # Creating a vector for the z-axis with swing height and commanded height.

38. return foot_location * np.array([1, 1, 0]) + z_vector + delta_foot_location # Returning the new foot location after applying position and height changes.

CMakeLists.txt:

Flowchart:


Set Minimum CMake Version
Define Project Name
Find Catkin Package
Set Up Python
Add Message Files
Generate Messages
Define Catkin Dependencies
Include Directories

This `CMakeLists.txt` file is used to configure a ROS package (`dingo_control`). It helps define how the package is built and linked, manages dependencies, and sets up installation and testing. Below is a detailed explanation of the key sections:

Basic Setup

cmake_minimum_required(VERSION 3.0.2)

project(dingo_control)

- This line specifies the minimum version of CMake required to build the package and names the project `dingo_control`.

Finding Dependencies

find_package(catkin REQUIRED COMPONENTS

  rospy message_generation std_msgs geometry_msgs

)

- This section tells CMake to find and include the `catkin` build system along with the required ROS dependencies:
    - `rospy`: Python interface to ROS.
    - `message_generation`: Used for generating ROS messages.
    - `std_msgs`: Standard message types like `String`, `Int32`, etc.
    - `geometry_msgs`: Message types for 3D geometry, such as `Point`, `Pose`, etc.

Python Setup

catkin_python_setup()

- This line indicates that the package uses Python, and if a `setup.py` script is present, it will ensure Python modules and global scripts are properly installed.

Message Generation

```
add_message_files(
  FILES
  TaskSpace.msg
  JointSpace.msg
  Angle.msg
)
```

- This section declares the message files (`TaskSpace.msg`, `JointSpace.msg`, `Angle.msg`) to be processed by `message_generation`. These are custom message types defined within the package.

```
generate_messages(
  DEPENDENCIES
  std_msgs geometry_msgs
)
```

- This line tells CMake to generate the necessary message files from the `.msg` files, and it also declares dependencies on `std_msgs` and `geometry_msgs`.

catkin Package Declaration

```
catkin_package(
  CATKIN_DEPENDS rospy message_runtime
```

)

- This macro declares that the package depends on `rospy` (for Python functionality) and `message_runtime` (for running the generated messages). It also manages the inclusion of headers, libraries, and dependencies.

Include Directories

include_directories(

  ${catkin_INCLUDE_DIRS}

)

- This line adds the directories where ROS and `catkin` headers are located so that the package can use them during compilation.

Executable and Library Setup (Commented Out)

add_library(${PROJECT_NAME} src/${PROJECT_NAME}/dingo_control.cpp)

add_executable(${PROJECT_NAME}_node src/dingo_control_node.cpp)

- These lines (commented out) would normally declare a C++ library or executable to be built. For example, `dingo_control_node.cpp` would be compiled as an executable if needed.

Install Setup (Commented Out)

install(TARGETS ${PROJECT_NAME}_node

RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}

)

- The `install()` commands are used to specify where to place the installed files. For example, executable nodes would be placed in the binary directory.

Testing Setup (Commented Out)

```
# catkin_add_gtest(${PROJECT_NAME}-test test/test_dingo_control.cpp)

# if(TARGET ${PROJECT_NAME}-test)

#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})

# endif()
```

- This section (also commented out) would be used to add Google Test-based C++ tests. If enabled, tests would be compiled and linked against the package.

```
# catkin_add_nosetests(test)
```

- This line (commented out) sets up a Python-based test system using `nosetests`. `catkin_add_nosetests(test)` would automatically find and run any Python tests under the `test` directory.

Summary

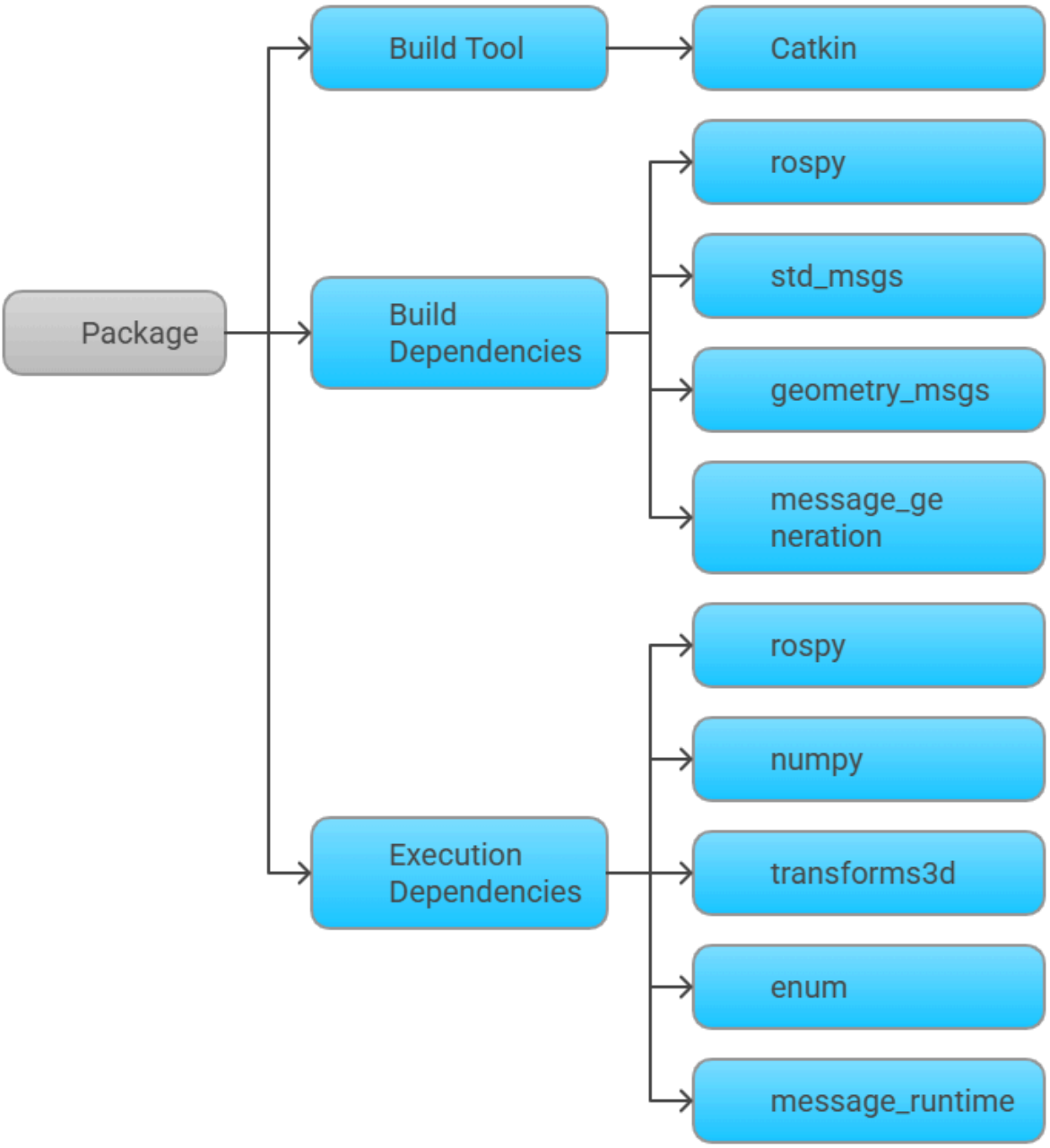
- **Message Generation**: Custom message types like `TaskSpace.msg` are generated using `message_generation`.
- **Dependencies**: This package depends on `rospy`, `std_msgs`, `geometry_msgs`, and `message_runtime`.
- **Executable Setup**: The package does not currently compile any executables or libraries (as the relevant lines are commented out).
- **Installation**: There are placeholders for installing executable scripts, binaries, and libraries, though they are commented out.

- **Testing**: Placeholder for both C++ (`gtest`) and Python (`nosetests`) tests, although not active in this configuration.

Commented Sections

- A lot of sections (e.g., `add_executable`, `install`, `catkin_add_gtest`) are commented out, which means they are not being used right now, but could be useful when the package needs to be extended.

Package.xml:



The `package.xml` file is a configuration file used by ROS (Robot Operating System) to define and manage metadata about a ROS package. It is essential for building and managing the package, specifying the package dependencies, licenses, maintainers, and other important information. This file is used by tools like `catkin`, `rosdep`, and the ROS build system to understand how to handle and integrate the package.

```
1  <?xml version="1.0"?>  <!-- Declares the version of XML being used, which is 1.0
-->
```

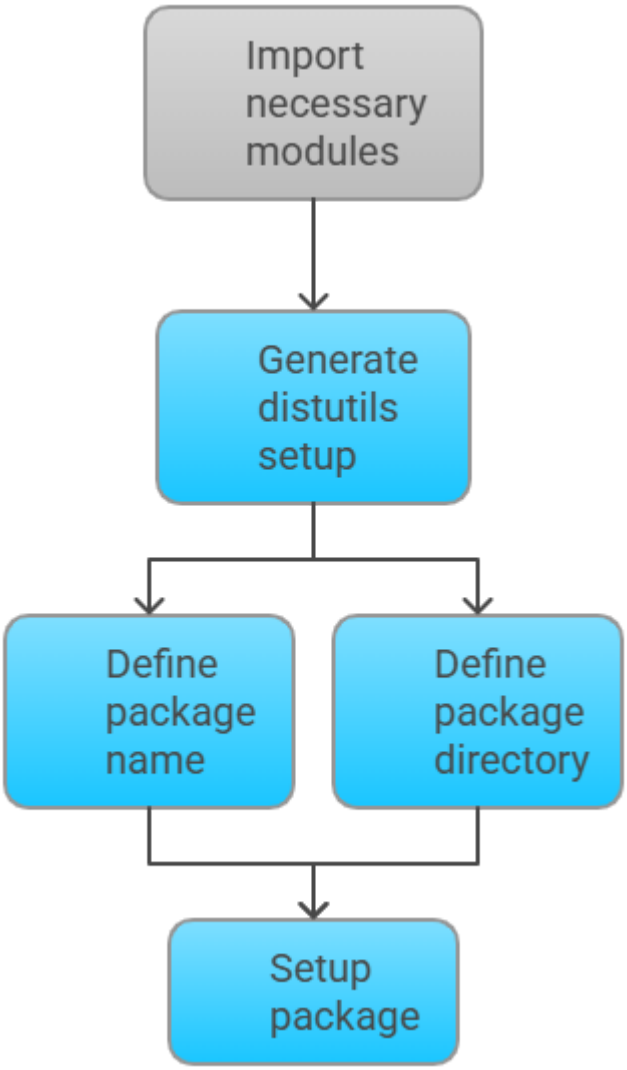
```
2  <package format="2">  <!-- Defines the start of the package declaration, format="2" specifies the XML format version -->

3    <name>dingo_control</name>  <!-- Specifies the name of the package -->

4

5    <version>0.0.0</version>  <!-- Specifies the version of the package (0.0.0 indicates it's in early development) -->

6    <description>The dingo_control package</description>  <!-- Provides a brief description of what the package does -->

7    <!-- Maintainer tag is required to specify who is responsible for maintaining the package -->

8    <!-- One maintainer tag is required, multiple allowed -->

9    <maintainer email="alex@todo.todo">alex</maintainer>  <!-- The maintainer of the package, with email contact -->

10   <!-- License tag is required to specify the licensing terms of the package -->

11   <!-- Common license options are BSD, MIT, GPL, etc. -->

12   <license>TODO</license>  <!-- The license hasn't been set yet; "TODO" is a placeholder -->

13   <!-- URL tags are optional, but can be used to specify external links -->

14   <!-- For example: <url type="website">http://wiki.ros.org/dingo_control</url> -->

15   <!-- Author tags are optional and can be used to list contributors or authors -->

16   <!-- Multiple author tags are allowed, though maintainers can also be authors -->

17   <!-- Example: <author email="jane.doe@example.com">Jane Doe</author> -->
```

```
18  <!-- The *depend tags are used to list dependencies that the package needs -->

19  <!-- Dependencies can be ROS packages or system packages -->


20  <!-- buildtool_depend: Specifies packages needed for the build system -->

21  <buildtool_depend>catkin</buildtool_depend>  <!-- Declares catkin as a build
tool dependency -->


22  <!-- build_depend: Declares dependencies that are needed at build time -->

23  <build_depend>rospy</build_depend>  <!-- Declares rospy (ROS Python library)
as a build-time dependency -->

24  <build_export_depend>rospy</build_export_depend>  <!-- Export the rospy
dependency to be available for other packages using this one -->


25  <build_depend>std_msgs</build_depend>  <!-- Declares std_msgs (ROS
standard messages) as a build-time dependency -->

26  <build_depend>geometry_msgs</build_depend>  <!-- Declares geometry_msgs
(ROS messages for geometric data) as a build-time dependency -->

27  <build_depend>message_generation</build_depend>  <!-- Declares
message_generation to generate messages, services, and actions -->


28  <!-- exec_depend: Declares dependencies required at runtime -->

29  <exec_depend>rospy</exec_depend>  <!-- Declares rospy as a runtime
dependency -->


30  <depend>numpy</depend>  <!-- Declares numpy (Python library for numerical
computations) as a runtime dependency -->

31  <depend>transforms3d</depend>  <!-- Declares transforms3d (Python library
for 3D transformations) as a runtime dependency -->
```

```
32   <depend>enum</depend>  <!-- Declares enum (Python library for
enumerations) as a runtime dependency -->


33   <exec_depend>message_runtime</exec_depend>  <!-- Declares
message_runtime (ROS package for handling generated messages) as a runtime
dependency -->


34   <!-- Export tag can be used to specify additional metadata for tools -->

35   <!-- For instance, this could specify ROS-specific configurations or additional
info -->

36   <export>

37     <!-- Other tools can request additional information be placed here -->

38   </export>


39 </package>  <!-- Closes the package definition -->
```

Setup.py:

Flowchart:



This Python script is used for setting up a ROS (Robot Operating System) Python package. It leverages `distutils` and `catkin_pkg` to facilitate packaging and distribution of the Python package. Specifically, this script is used to generate a `setup.py` file that ROS uses for building and installing the Python package.

1 `from distutils.core import setup` # Import the 'setup' function from the distutils module, which is used to package Python code for distribution.

2 `from catkin_pkg.python_setup import generate_distutils_setup` # Import the 'generate_distutils_setup' function from the catkin_pkg package. This utility simplifies the creation of the distutils setup for ROS packages.

3 d = generate_distutils_setup(# Call the 'generate_distutils_setup' function to generate the arguments needed for the 'setup' function.

4 packages=['dingo_control'], # Specifies the list of Python packages to include. In this case, it is 'dingo_control', which is the package we are setting up.

5 package_dir={'': 'src'} # Specifies the directory structure for the package. Here, it tells ROS that the source code is located under the 'src' folder.

6) # End of the 'generate_distutils_setup' function call. The return value is stored in the variable 'd'.

7 setup(**d) # Calls the 'setup' function from distutils. It unpacks the dictionary 'd' (which contains the package setup details) and passes it as keyword arguments to the 'setup' function.

Purpose of the Script:

- **ROS Python Package Setup**: In ROS, the `setup.py` file is used to configure how the Python package is built, installed, and distributed. This script automates the process by using the `generate_distutils_setup` function from `catkin_pkg`, which ensures that the setup is ROS-compatible.

- **Simplification**: The `generate_distutils_setup` function simplifies the setup process by automatically detecting package structure and generating the necessary `setup()` arguments.

Why is this script used?

- **Packaging**: ROS packages that contain Python code need a `setup.py` file to be properly built and installed. This script ensures that the `dingo_control` package, located in the `src` directory, is properly set up for distribution and use within the ROS ecosystem.

- **ROS Integration**: The use of `generate_distutils_setup` ensures the proper setup for the ROS build system (Catkin), which can handle Python packages and their dependencies.