Create3 Random Walk:
Going on a Hunt
Robotics I Spring 2023

Jordan Reed

February 28, 2023

Introduction

The following is a description of an example of a random walk program implemented using ROS2 for the Create3 robot, affectionately nicknamed 'Monster'.

Block Diagram

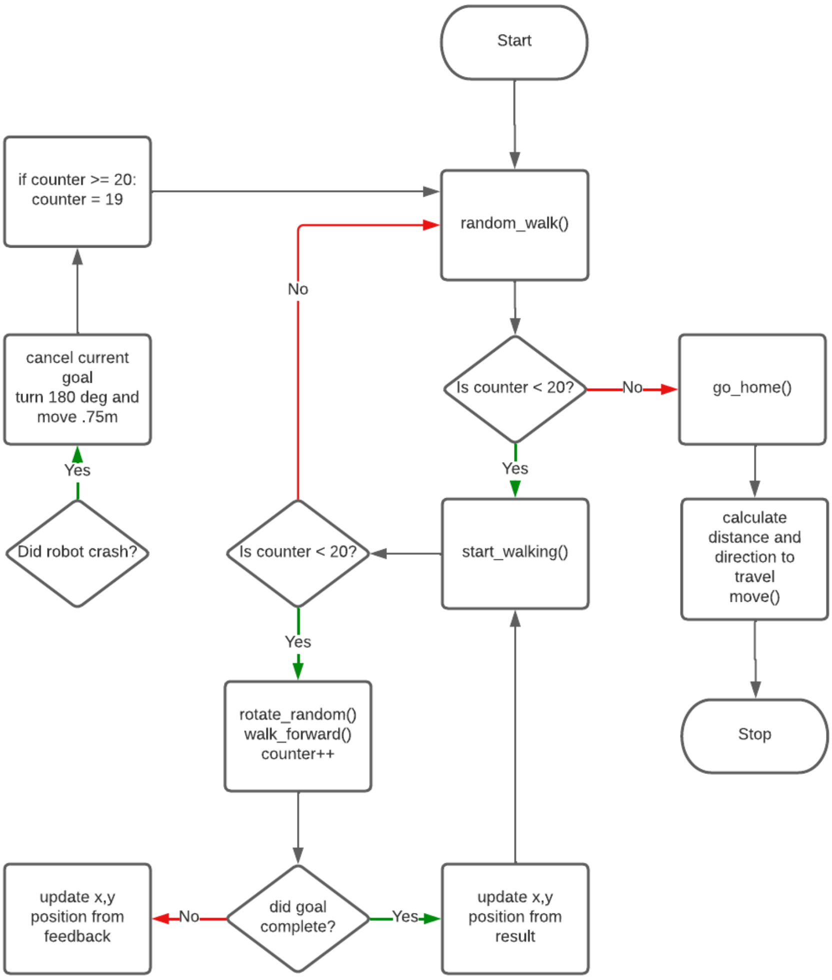

Figure 1: Block Diagram

Basic Program Loop

The program starts by undocking, then moving forward 1 meter. After starting, the program will enter into the random_walk() loop.

In the random_walk(), the program will create and send goals to rotate a random direction, and move forward .75 meters It will then increment the counter. After 10 loops of this, the program will initiate the go_home() sequence. The go_home() sequence will calculate the direction and distance the robot needs to go in order to make it back to the dock, then execute those commands.

Running in a separate thread throughout the entire program is a subscriber. The subscriber will call a function to handle any bump detections. If a bump is detected, the program will cancel the current goal, and move the robot backward, before resuming it's regular program.

If the robot crashes on the way back home, the listener function will decrement the counter for the random_walk(). This will allow the robot to, hopefully, move past the obstacle and resume going home.

The specifics of the algorithms are discussed below.

Algorithm Descriptions

Node Creation

The entire robot movement is encapsulated in a class inherited from rclpy.node, nicknamed 'Monster'. This class handles the movement functions as well as keeping track of the different variables needed for running the program. These variables include: current goal id (goal_uuid), x and y position, the robot rotation (quatz), the random iteration counter (counter), and a single action client (action_client) that is reused for each goal sent to the robot.

The node also sets up a subscriber, which listens for hazards the robot runs into. This is handled in a listener callback, which is described below.

When the node is initialized, it also sets up 2 mutually exclusive callback groups to handle the actions sent to the robot and the subscriber/listener actions.

Send Generic Goal

This program has a function that handles sending a goal to the robot. This is set up so you can send any goal you need through one function, instead of rewriting the same code over and over for slightly different goals.

This function will send a goal asynchronously to the robot, while in a lock. This ensures that the program is only sending one goal at a time. Instead of adding callbacks, the function will enter into a while loop until the goal has been sent. At this point, the goal_uuid is grabbed in case we need to cancel it. Again, the function will enter into a loop until the goal has been accepted, then again when it is done. This could be done with callbacks, but there were significantly fewer issues when this was handled in the function in loops. After the function has sent the goal and it's been completed, the function will reset the goal_uuid variable while in a lock. This helps make sure that another thread/function is not trying to access the goal_uuid at the same time.

This function does use a get_result_callback, described below.

Get Result Callback

Sending a goal to the robot will also spawn a get_result_callback. This callback is used to get the result object after a goal has been completed. The result object contains the information on position and rotation of the robot. This allows us to keep track of where the robot is compared to its dock so we can accurately send it home at the end of its random walk.

Listener Callback

There is a listener callback that called whenever the subscriber receives a message to the 'hazard_detection' topic. The callback will parse through the message and see if it is a bumper response. If it is a bumper response, the callback acquires the lock, and cancels the current goal, whatever it is. The goal_uuid is cleared and the callback initiates the back_up() sequence once the lock is released. However, if the robot crashes on it's way home, the counter variable is reset so the random_walk() function will reengage one more time. This is to allow the robot to try and avoid the obstacle in between itself and its home.

Back Up

When the robot has hit an obstacle, the listener callback will call back_up(). This function will send a rotate 180°goal, then a move forward .75 meters goal.

Random Walk

The random_walk() function is the function that handles most of the logic for the program. This function will loop through 10 times (using the counter variable) and send a RotateAngle goal (with a random angle from 0-360°) and a drive forward .75 meters goal. It will then increment the counter variable from inside a lock to ensure there are no deadlocks. If the program has looped 10 times, the program will start the go_home() sequence.

Go Home

This function calculates the right rotation and distance for the robot to travel in order to maneuver into the right position to start docking. At least, that was the original idea. It currently is calling the NavigateToPosition action supplied by the irobot_create_msgs. The Create3 robots do actually keep track of their relative position, so as long as the program is started from the dock every time, this action should work.

Main

The main function handles initializing rclpy, the MultiThreadedExecutor (for multiple threads), and the node. The node's functions are implemented through a callback system using Garrett's KeyCommander code. When the correct key is pressed, the program starts.

Conclusion/Results

I initially had set up the program to run all actions asynchronously, and to spin on the node only when an action was sent. However, in trying to integrate my code with the example code, I ran into issues with the MultiThreadedExecutor. I opted to, instead, adopt the way of the multi-threaded skeleton example provided. The program works, most of the time. There seem to be issues when the Create3 robot gets stuck too much, such as wedged between a door and a wall.

Source Code

I've included the complete main.py file for reference. As the KeyCommander code was given in class, I did not include it here.

```python
import rclpy
from rclpy.node import Node
from rclpy.action.client import ActionClient
from rclpy.qos import qos_profile_sensor_data
from rclpy.callback_groups import MutuallyExclusiveCallbackGroup
from geometry_msgs.msg import PoseStamped
from action_msgs.msg._goal_status import GoalStatus

import irobot_create_msgs
from irobot_create_msgs.action import DriveDistance, Undock, RotateAngle, Dock, NavigateToPosition
from irobot_create_msgs.msg import HazardDetectionVector

from pynput.keyboard import KeyCode
from key_commander import KeyCommander
from threading import Lock
from rclpy.executors import MultiThreadedExecutor

import random, math, time


# To help with Multithreading
lock = Lock()

class Monster(Node):
    """
    Class to coordinate actions and subscriptions
    """

    def __init__(self, namespace):
        super().__init__('monster_hunting')

        # 2 Seperate Callback Groups for handling the bumper Subscription and Action Clients
        cb_Subscripion = MutuallyExclusiveCallbackGroup()
        self.cb_action =MutuallyExclusiveCallbackGroup()

        # Subscription to Hazards, the callback function attached only looks for bumper hits
        self.subscription = self.create_subscription(
            HazardDetectionVector, f'/{namespace}/hazard_detection', self.listener_callback, qos_profile_sensor_data,callback_group=cb_Subscripion)

        self._action_client = None          # reuse action client
        self._namespace = namespace

        # Variables
        self._goal_uuid = None          # for goal handling in mult. threads

        self.counter = 0                # for random walk iteration counting

        self.x = 0                      # x position
        self.y = 0                      # y position
        self.quatz = 0                  # rotation of robot

    # -------------------------------------------------------------------------
    # generic action client and node functions

    def listener_callback(self, msg):
        '''
        This function is called every time self.subscription gets a message
        from the Robot. Here it parses the message from the Robot and if its
        a 'bump' message, cancel the current action.
        '''

        # If it wasn't doing anything, there's nothing to cancel
        if self._goal_uuid is None:
            return

        # msg.detections is an array of HazardDetection from HazardDetectionVectors.
        # Other types can be gotten from HazardDetection.msg
        for detection in msg.detections:
            if detection.type == 1:   #If it is a bump
                self.get_logger().warning('HAZARD DETECTED')

                with lock: # Make this the only thing happening
                    self.get_logger().warning('CANCELING GOAL')
                    self._goal_uuid.cancel_goal_async()

                    # Loop until the goal status returns canceled
                    while self._goal_uuid.status is not GoalStatus.STATUS_CANCELED and self._goal_uuid.status is not None:
                        pass

                    self.get_logger().info("Goal canceled")

                    if self.counter >= 10:
                        self.counter = 9

                    print("sleeping before starting back up")
                    time.sleep(2)

                print("calling back up sequence")
                self.back_up()

    def send_generic_goal(self, action_type, action_name:str, goal):
        """
        function to send any goal async

        :param action_type: imported from irobot_create_messages
        :param action_name: string that is used with namespace
        :param goal: the goal object for action
        """
        print("\n")
        self.get_logger().info(f"Sending goal for '{action_name}'")
        # create action client with goal info
        self._action_client = ActionClient(self, action_type, f'/{self._namespace}/{action_name}', callback_group= self.cb_action)

        # wait for server
        self.get_logger().warning("Waiting for server...")
        self._action_client.wait_for_server()

        # server available
        self.get_logger().warning("Server available. Sending goal now...")

        # send goal in a lock and wait for send to finish
        with lock:
            send_goal_future = self._action_client.send_goal_async(goal)
            while not send_goal_future.done():
                pass

            # set goal uuid so we know we have a goal in progress
            self._goal_uuid = send_goal_future.result()

        while self._goal_uuid.status == GoalStatus.STATUS_UNKNOWN:
            # wait until status is set to something
            pass

        self.get_logger().warning("Goal is in progress")
        while self._goal_uuid.status is not GoalStatus.STATUS_SUCCEEDED:
            if self._goal_uuid.status is GoalStatus.STATUS_CANCELED:
                break # If the goal was canceled, stop looping otherwise loop till finished
            pass

        self.get_logger().info("Goal completed!")

        # get result object for positioning
        get_result_future = self._goal_uuid.get_result_async()
        get_result_future.add_done_callback(self.get_result_callback)

        # reset goal uuid to none after action is completed
        with lock:
            self._goal_uuid = None

        # goal completed
        self.get_logger().warning(f"{action_name} action done")

    def get_result_callback(self, future):
        """
        a callback that will grab the position out of the result

        :param future: future passed in for get result
        """
        result = future.result().result

        # get updated position and rotation
        try:
            pos = result.pose
            self.x = pos.pose.position.x # needed
            self.y = pos.pose.position.y # needed

            self.quatz = pos.pose.orientation.z # needed

            print(f'current position: ({self.x:.3f}, {self.y:.3f}')
            print(f'current rotation: ({self.quatz:.3f})')
        except Exception as e:
            pass

    # -------------------------------------------------------------

    def start_hunting(self):
        """
        runs start up then start the random walk
        """

        self.start_up()
        self.random_walk()

    def start_up(self):
        """
        undock and then move forward 1m
        """
        # undock
        goal = Undock.Goal()
        self.send_generic_goal(Undock, 'undock', goal)

        # drive forward 1m
        goal = DriveDistance.Goal()
        goal.distance = 1.0
        self.send_generic_goal(DriveDistance, 'drive_distance', goal)

    def random_walk(self):
        """
        robot goes through loop of : rotate random angle, move forward .75m.
        If 10 iterations have been done, go home
        """
        max_iter = 10
        while self.counter < max_iter:
            print(f'in random walk loop counter: {self.counter}')

            # rotate random
            rand_angle = random.randrange(0,360)
            rand_angle = math.radians(rand_angle)
            cur_goal = RotateAngle.Goal()
            cur_goal.angle = rand_angle # from 0 to 360 deg
            self.send_generic_goal(RotateAngle, 'rotate_angle', cur_goal)

            # walk forward .75m
            cur_goal = DriveDistance.Goal()
            cur_goal.distance = .75
            self.send_generic_goal(DriveDistance, 'drive_distance', cur_goal)

            with lock:
                print("in lock for incrementing counter")
                self.counter += 1

        if self.counter >= max_iter:
            # go home
            print("monster is ready to go home now")
            self.go_home()

    def go_home(self):
        print("monster is looking for home")

        try:
            goal = NavigateToPosition.Goal()
            goal.goal_pose = PoseStamped(0, .6)
            self.send_generic_goal(NavigateToPosition, "navigate_to_position", goal
    )
        except Exception as e:
            print(f'error with navigate: {e}')

        goal = Dock.Goal()
        self.send_generic_goal(Dock, 'dock', goal)

    def back_up(self):
        """
        called in the cancel callback
        rotate 180 deg and move .75m to go backward
        """
        self.get_logger().info('Starting back up process...')

        # rotate 180 deg
        goal = RotateAngle.Goal()
        goal.angle = math.pi # 180 deg
        self.send_generic_goal(RotateAngle, 'rotate_angle', goal)

        # walk forward .75m
        goal = DriveDistance.Goal()
        goal.distance = .75
        self.send_generic_goal(DriveDistance, 'drive_distance', goal)


if __name__ == '__main__':
    rclpy.init()

    namespace = 'create3_0620'
    m = Monster(namespace)

    # 1 thread for the Subscription, another for the Action Clients
    exec = MultiThreadedExecutor(2)
    exec.add_node(m)

    keycom = KeyCommander([
        (KeyCode(char='r'), m.start_hunting),
        ])

    print("r: Start hunting")
    try:
        exec.spin() # execute slash callbacks until shutdown or destroy is called
    except KeyboardInterrupt:
        print('KeyboardInterrupt, shutting down.')
        print("Shutting down executor")
        exec.shutdown()
        print("Destroying Monster Node")
        m.destroy_node()
        print("Shutting down RCLPY")
        rclpy.try_shutdown()
```