Controller

Well done on making it to Step 2 of Task 1B! So now that we've,

- created an urdf file,
- launched gazebo, spawned the robot and
- are able to actuate it manually

In step 2, Let's automate the robot, by creating a controller rospy node that will make the robot automatically go to desired goal pose (pose, refers to the position AND orientation of the robot).

Introduction

- go-to-goal controller
  - given ideal localisation and simplified kinematics from step 1.

skipping/simplified localisation: The **odom** topic (as declared in the urdf file) is directly giving us the present pose of the robot. Which is not the case in real world. We will eventually use a marker and openCV to localise the robot in a more realistic way. [localise: to answer "Where the Robot is?", specifically, what is the present (x, y, theta) of the robot.]

skipping/simplified kinematics: The controller we are designing, outputs something like `[v_x, v_y, omega]` that is sent to cmd_vel, which independently controls the velocities of chassis directly. But in hardware, we are not directly actuating the chassis by giving `[v_x, v_y, w]`, but we give wheel velocities, which is `[v_1, v_2, v_3]` for a three omni-wheeled robot. We are skipping the problem of finding `[v_1, v_2, v_3]` given `[v_x, v_y, w]`.

These topics will be covered in Task 2

So what is Holonomic Drive?

- An object in the physical "3D" space we live in has 6 Degreed of Freedom.

  - 3 translations, 3 rotations.

- An object on a "2D" plane has 3 Degreed of freedom:

  - 2 translations and 1 rotation.
  - In our convention, it will be translation in the X and Y axis and rotation about the Z axis.

- **Ground vehicles live on a plane.**
  The popular differential drive robot has a Non Holonomic CONSTRAINT. That doesn't allow the robot to translate in one of the axis (say X). This is a constraint on velocity and NOT a constraint on position. So although a differential drive CAN parallel park it has to make many complex manoeuvres to achieve it. While a holonomic drive (ex: Omni wheel robot) can simply translate in that direction (X) since it has no such constraint in that direction.

- I.E. The ground vehicle can directly control velocities in ALL the 3 Degrees of Freedom possible. I.E. Control [v_x, v_y, w] linear velocity in X-Y and Omega: angular velocity in the Z axis. (Unlike only two, [v, w] (or [v_y, w] in our convention) for a differential drive robot.)

The above block of explanation might now give you some clarity on what is holonomic drive.

This leaves one more major question unanswered: What is Forward and Inverse Kinematics? In our specific case of **three-omni-wheeled robot**, We could simplify it by saying, it's the relationship between **[v_1, v_2, v_3]** and **[v_x, v_y, w]** That's all for now, more about this in Task 2!

Enough chit-chat, let's get down to business!

The Task

So continuing from where we left off in Step1. We now have a launch file which opens gazebo, empty world and spawns the robot.

If you do that and then do rostopic list, you should find two topic of interest:

- **/cmd_vel**
- **/odom**

which are defined in the urdf file, gazebo plugin.

Now we shall create a rospy node: controller.py that will

- **subscribe to /odom** and
- **publish to /cmd_vel**

1. So in your package directory, create a **your_package_name/scripts/** directory and create a file name controller.py inside it.

2. Now let's start writing the **controller.py** file.

We'll need to **import** the following modules:

```
#!/usr/bin/env python3
import rclpy                                                # ROS 2 Python library for creating
from rclpy.node import Node                                 # Node class for creating ROS 2 noc
from geometry_msgs.msg import Twist                         # Publishing to /cmd_vel with msg t
from nav_msgs.msg import Odometry                           # Subscribing to /odom with msg typ
import time                                                 # Python time module for time-relat
import math                                                 # Python math module for mathematic
from tf.transformations import euler_from_quaternion       # Odometry is given as a quaternior
from my_robot_interfaces.srv import NextGoal               # Service
```

We'll need some variables to keep track of **pose** of the robot, **x, y, theta**.

```
self.hb_x = 0
self.hb_y = 0
self.hb_theta = 0
```

We'll need a callback function for subscribing to **/odom**. As you must be aware by now, this function will be automatically called everytime to update the pose of the robot (whenever there is an update in the `/odom` topic).

```
def odometryCb(msg):
    global hb_x, hb_y, hb_theta

    # Write your code to take the msg and update the three variables
```

ROS2 Odometry documentation is given **here** for reference.

Have a look at the data structure of the msg received as argument (Odometry message) and figure out how to get `hb_x`, `hb_y` and `hb_theta` from that.

Hint: You only need to look at the pose part of the data.

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import time
import math
from tf_transformations import euler_from_quaternion
from my_robot_interfaces.srv import NextGoal

class HBTask1BController(Node):

   def __init__(self):
        super().__init__('hb_task1b_controller')

        # Initialze Publisher and Subscriber
        # We'll leave this for you to figure out the syntax for
        # initialising publisher and subscriber of cmd_vel and odom respectively

        # Declare a Twist message
        self.vel = Twist()
        # Initialise the required variables to 0

        # For maintaining control loop rate.
        self.rate = self.create_rate(100)
        # Initialise variables that may be needed for the control loop
        # For ex: x_d, y_d, theta_d (in **meters** and **radians**) for defining desired
        # and also Kp values for the P Controller


        # client for the "next_goal" service
        self.cli = self.create_client(NextGoal, 'next_goal')
        self.req = NextGoal.Request()
        self.index = 0

   def main(args=None):
    rclpy.init(args=args)

    # Create an instance of the EbotController class
    ebot_controller = HBTask1BController()

    # Send an initial request with the index from ebot_controller.index
    ebot_controller.send_request(ebot_controller.index)

    # Main loop
    while rclpy.ok():

        # Check if the service call is done
        if ebot_controller.future.done():
            try:
                # response from the service call
                response = ebot_controller.future.result()
            except Exception as e:
                ebot_controller.get_logger().infselfo(
                    'Service call failed %r' % (e,))
            else:
                #########           GOAL POSE             #########
                x_goal      = response.x_goal
                y_goal      = response.y_goal
                theta_goal  = response.theta_goal
                ebot_controller.flag = response.end_of_list
                ####################################################

                # Find error (in x, y and theta) in global frame
                # the /odom topic is giving pose of the robot in global frame
                # the desired pose is declared above and defined by you in global frame
                # therefore calculate error in global frame

                # (Calculate error in body frame)
                # But for Controller outputs robot velocity in robot_body frame,
                # i.e. velocity are define is in x, y of the robot frame,
                # Notice: the direction of z axis says the same in global and body frame
                # therefore the errors will have have to be calculated in body frame.
                #
                # This is probably the crux of Task 1, figure this out and rest should be

                # Finally implement a P controller
                # to react to the error with velocities in x, y and theta.

                # Safety Check
                # make sure the velocities are within a range.
                # for now since we are in a simulator and we are not dealing with actual
                # we may get away with skipping this step. But it will be very necessary


                #If Condition is up to you

                ############     DO NOT MODIFY THIS       #########
                ebot_controller.index += 1
                if ebot_controller.flag == 1 :
                    ebot_controller.index = 0
                ebot_controller.send_request(ebot_controller.index)
                ####################################################

        # Spin once to process callbacks
        rclpy.spin_once(ebot_controller)

    # Destroy the node and shut down ROS
    ebot_controller.destroy_node()
    rclpy.shutdown()

    if __name__ == '__main__':
        main()
```

Initially, if you feel that using services is too overwhelming, you can use the co-ordinates shared below to check if your bot is making a square or not.
Do note that this is only for testing purpose and will not be considered for task submission.

```
    x = [4, -4, -4, 4, 0]
    y = [4, 4, -4, -4, 0]
    theta = [0, 0, 0, 0, 0]
    # the following theta desired are optional since throughout hologlyph theme we sh
    # theta = [0, PI/2, PI, -PI/2, 0]
```

Click here for the data structure of the **Twist message** to figure out how to initialise and set x, y linear velocity and z angular velocity.

Finally we come to exciting part of the code...

Control Loop!

```
while rclpy.ok():

    # Find error (in x, y and theta) in global frame
    # the /odom topic is giving pose of the robot in global frame
    # the desired pose is declared above and defined by you in global frame
    # therefore calculate error in global frame

    # (Calculate error in body frame)
    # But for Controller outputs robot velocity in robot_body frame,
    # i.e. velocity are define is in x, y of the robot frame,
    # Notice: the direction of z axis says the same in global and body frame
    # therefore the errors will have have to be calculated in body frame.
    #
    # This is probably the crux of Task 1, figure this out and rest should be fine.

    # Finally implement a P controller
```

```
        # to react to the error with velocities in x, y and theta.

        # Safety Check
        # make sure the velocities are within a range.
        # for now since we are in a simulator and we are not dealing with actual physical
        # we may get away with skipping this step. But it will be very necessary in the l

        rclpy.spin_once(ebot_controller)

    ebot_controller.destroy_node()
    rclpy.shutdown()
```

The content of the comment in above code very import so let's repeat it here:

- Find error (in x, y and theta) in global frame
    - the /odom topic is giving present pose of the robot in global frame
    - the desired pose is declared above and defined by you in global frame therefore calculate error in global frame
- Calculate error in body frame
    - Controller outputs robot velocity in robot_body frame, i.e. velocity are define is in x, y of the robot frame,
      Notice: the direction of z axis says the same in global and body frame therefore the
- Finally implement **P controllers** to react to the error in robot_body frame with velocities in x, y and theta in robot_body frame: [v_x, v_y, w]

❗ This is probably the crux of Task 1B
Please note this is just the approach we took, there may be other approaches that you may come up with.
Feel absolutely free to go ahead with that approach.

The **objective** is simple
Make the robot go to desired [x_d, y_d, theta_d] by using feedback of present [x, y, theta].

Thats it!

If the robot goes to the desired goal pose (defined by you), Congratulations! You have achieved a major milestone!

Now it's time to extend (add some logic) the above code to handle a **sequence** of desired poses.

So now [x_d, y_d, theta_d] instead of being single values we shall have a list of desired goal poses.

```
# Example
self.x_goals = [1, -1, -1, 1, 0]
self.y_goals = [1, 1, -1, -1, 0]
self.theta_goals = [0, 0, 0, 0, 0]
```

Once you have entered a list of goal poses, make necessary changes to the control loop to:

- go-to-goal-pose of a certain index (index = 0 at start)
- identify if the goal has been reached (write an if condition)
- stabilise/stay at the goal pose for at least 1 second
- increment index if index < length of the list.
- repeat

That's it!

If the robot goes to the sequence of desired goal poses one after other (defined by you), Congratulations you are almost done with Task 1!🥳