TURTLE_TRAJECTORIES

Automated Turtle Movement and Shape Drawing in ROS

BY:

BELAL EDOOR

MOHAMMAD ALHOOR

MAY 20, 2025

PALESTINE POLYTECHNIC UNIVERSITY
HEBRON

Introduction

This project provides a Python script using the Robot Operating System (ROS) to control the TurtleSim robot. The turtle can be commanded to follow various motion trajectories such as drawing geometric shapes (square, triangle, circle, hexagon), a spiral, a sine wave, or moving from one point to another.

Features

- Draw geometric shapes: square, triangle, circle, hexagon
- Create advanced patterns: spiral motion, sine wave path
- Point-to-point navigation
- Reset simulation environment
- Boundary checks to prevent turtle from exiting the window

Project Structure

- turtle_controller.py: Main Python script that implements all movement logic
- Uses:
  - /turtle1/cmd_vel topic to publish velocity commands
  - /turtle1/pose topic to subscribe to current turtle pose
  - /reset service to reset simulation

Project's files

1. Launch File:

```xml
<launch>
  <node pkg="turtle_trajectories" type="turtle_trajectories.py" name="publisher_node" output="screen" />
  <node pkg="turtlesim" type="turtlesim_node" name="sim" output="screen" />
</launch>
```

This ROS launch file is designed to initiate both the simulation environment and a custom control node in one step. Specifically, it starts the standard turtlesim_node, which opens a graphical window representing the turtle's world, and simultaneously launches the turtle_trajectories.py script from the turtle_trajectories package. This script contains the logic for controlling the turtle's movement, such as drawing geometric shapes or following complex paths. By including both nodes in a single launch file, users can streamline the testing and demonstration process. This setup is especially useful for educational purposes or development workflows where automated, reproducible turtle behaviors are needed within the turtlesim environment.

Turtle trajectorieas.py file

This Python script for ROS (Robot Operating System) is designed to control a turtle within the `turtlesim` simulator by executing a variety of movement patterns. The script starts by initializing ROS nodes and subscribing to the turtle's position updates while publishing movement commands through the `geometry_msgs/Twist` topic. It includes multiple user-defined functions to move the turtle in different geometric trajectories such as squares, triangles, circles, hexagons, and advanced patterns like spirals, sine waves, and direct point-to-point navigation.

Each shape or pattern is generated by combining linear and angular velocity commands, calculated using geometric relationships and time-based estimations. The script ensures that the turtle remains within the simulation window by predicting future positions and applying boundary checks. If a movement would take the turtle outside the visible area, the motion is stopped to avoid errors. Additionally, the script interacts with ROS services to reset the simulation environment when needed, allowing for clean reruns of each motion pattern.

The user is guided through a menu in the terminal, where they can input desired parameters such as side lengths, rotation angles, and direction of motion. This interactive design makes the script a useful learning tool for understanding robot motion control, ROS message communication, and safe navigation in a simulated environment.

Turtlesim Linear Motion Controller with Boundary Check

```python
#!/usr/bin/env python3
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from std_srvs.srv import Empty
import math
import sys

pose = Pose()

def update_pose(data):
    global pose
    pose = data

def is_within_bounds(x, y, margin=1.0):
    return margin <= x <= 11 - margin and margin <= y <= 11 - margin

def move_linear(velocity_publisher, speed, distance, is_forward):
    #Create a new Twist message: This is the message that contains the movement commands (linear and angular).
    vel_msg = Twist()
    #Linear velocity (on the x-axis), Absolute value of velocity (always positive), The velocity is positive (forward), The speed is negative (backward).
    vel_msg.linear.x = abs(speed) if is_forward else -abs(speed)
    #There is no rotation during movement, only straight movement.
    vel_msg.angular.z = 0
    #Returns the current time as the reference point for calculating elapsed time.
    t0 = rospy.Time.now().to_sec()
    #Initialize a variable to track the distance traveled so far.
    current_distance = 0
    #Store the turtle's starting position.
    start_x = pose.x
    start_y = pose.y

    rate = rospy.Rate(50)
    #The loop continues as long as we haven't covered the required distance yet.
    while current_distance < distance:
        future_x = pose.x + math.cos(pose.theta) * 0.1
        future_y = pose.y + math.sin(pose.theta) * 0.1
        #Check: Is the turtle about to go out the window (out of bounds)?
        if not is_within_bounds(future_x, future_y):
            rospy.logwarn("Robot stopped: reached boundary of the turtlesim window.")
            break
        #Sends the current speed command to the turtle to move.
        velocity_publisher.publish(vel_msg)
        #Current time.
        t1 = rospy.Time.now().to_sec()
        #How many meters have you traveled? Using speed and time: Distance = Speed × Time.
        current_distance = speed * (t1 - t0)
        rospy.loginfo(f"Distance traveled: {current_distance:.2f}")

        rate.sleep()
    #Stop the turtle from moving completely. Send a Twist message at 0 velocity to stop it.
    vel_msg.linear.x = 0
    velocity_publisher.publish(vel_msg)
```

This ROS Python script is responsible for controlling the turtle's motion within the turtlesim simulation environment. It achieves this by subscribing to the turtle’s real-time position updates through the `Pose` message and issuing movement commands using the `Twist` message. One of the core components of the script is the `move_linear` function, which directs the turtle to move either forward or backward for a user-defined speed and distance. During this motion, the function continuously checks whether the turtle remains within the visible bounds of the simulator window. If the turtle approaches the boundary, it automatically stops to prevent exiting the simulation area. Instead of using position deltas alone, the script relies on a time-based approach—calculating the traveled distance using the speed and elapsed time—to manage motion duration accurately. This design ensures consistent performance and makes the turtle's path more predictable, even with slight variations in sensor update timing.

Rotate method

```python
locity_publisher: The same publisher that sends movement commands.
# angular_speed_degree: The angular velocity, but in degrees.
# angle_degree: The angle the turtle must turn (in degrees).
# clockwise: Rotation type: If True → clockwise.
def rotate(velocity_publisher, angular_speed_degree, angle_degree, clockwise):
    vel_msg = Twist()

    #abs(angular_speed_degree): Takes the absolute value of the speed (make sure it's positive).
    #math.radians(): Converts the speed from degrees to radians, because ROS uses radians for rotation.
    #Example:
    #ex.  0° → π/2 = 1.57 radians
    angular_speed = math.radians(abs(angular_speed_degree))

    # vel_msg.angular.z: The angular velocity about the z-axis (i.e., the turtle rotates).
    # If clockwise: Make it negative → -angular_speed.
    # If counter-clockwise: Keep it positive.
    vel_msg.angular.z = -angular_speed if clockwise else angular_speed

    t0 = rospy.Time.now().to_sec()

    #How many angles has the turtle turned so far?
    current_angle = 0
    rate = rospy.Rate(50)

    # We convert the angle from degrees to radians because all angular operations in ROS are based on radians.
    # We continue repeating as long as the turtle has rotated the desired angle.
    while current_angle < math.radians(angle_degree):

        #We send the current rotation command to the turtle.
        velocity_publisher.publish(vel_msg)
        t1 = rospy.Time.now().to_sec()

        # We calculate the angle it has turned so far using:
        # Angle = Angular Velocity × Time
        current_angle = angular_speed * (t1 - t0)
        rate.sleep()

    # top the rotation by setting the angular velocity to 0.
    # We send the message to make the turtle stop moving.
    vel_msg.angular.z = 0
    velocity_publisher.publish(vel_msg)
```

This code rotates the turtle in the ROS turtlesim environment by a user-specified angle at a chosen angular speed. It first converts the speed from degrees to radians, as ROS requires angular velocities in radians per second. The direction of rotation—clockwise or counterclockwise—is determined by the `clockwise` parameter, which sets the sign of the angular velocity. A loop is used to continuously publish the rotation command until the turtle completes the desired angle of rotation based on time calculations. After reaching the target angle, the turtle is stopped by sending a zero angular velocity command.

Draw square

```python
def draw_square(velocity_publisher):
    length = float(input("Enter the edge length of the square (<= 4.0): "))
    if length > 4.0:
        rospy.logwarn("Too large! Reducing to 4.0")
        return
    if length <= 0:
        rospy.logwarn("Length must be positive! Setting to 1.0")
        return
    # We repeat 4 times (because the square has 4 sides).
    for _ in range(4):

        # Calls the move_linear function.
        move_linear(velocity_publisher, 1.0, length, True)

        #Call the rotate function to rotate the turtle 90 degrees
        rotate(velocity_publisher, 30, 90, False)
```

This function makes the turtle draw a square by prompting the user to enter the edge length. It ensures the length is valid by limiting it to a range between 0 and 4.0 units. The turtle then performs a loop four times, each time moving forward by the given length and rotating 90 degrees. It uses the move_linear and rotate functions to handle straight movement and turning, respectively. This setup allows for flexible, safe square drawing based on user input.

Draw tringle

```python
def draw_triangle(velocity_publisher):
    side = float(input("Enter triangle side length (<= 4.0): "))
    if side > 4.0:
        rospy.logwarn("Too large! Reducing to 4.0")
        return

    if side <= 0:
        rospy.logwarn("Length must be positive! Setting to 1.0")
        return

    #We repeat the two steps (move + rotate) 3 times because the triangle has 3 sides.
    for _ in range(3):
        #Straight line motion is required.
        move_linear(velocity_publisher, 1.0, side, True)
        rotate(velocity_publisher, 30, 120, False)
```

This function instructs the turtle to draw an equilateral triangle. It starts by taking the side length from the user and ensures the value is within the safe range ($0 < \text{side} \leq 4.0$). Then, it loops three times once for each side of the triangle. In each loop, the turtle moves forward the given distance and then rotates 120 degrees to form the triangle's internal angles. It combines straight motion with angular rotation to trace the triangle accurately.

Draw circle

```python
def draw_circle(velocity_publisher):
    radius = float(input("Enter circle radius (<= 4.0): "))
    if radius > 4.0:
        rospy.logwarn("Too large! Reducing to 4.0")
        return
    if radius <= 0:
        rospy.logwarn("Radius must be positive! Setting to 1.0")
        return
    # The user is asked to specify the direction
    # 1 = clockwise.
    # 2 = counterclockwise.
    # strip() removes extra spaces (if the user presses Space by mistake).
    direction = input("Choose direction: 1 for clockwise, 2 for counter-clockwise: ")
    # If the user enters anything other than 1 or 2.
    # Prints an alert.
    # Resets the orientation to the default (counterclockwise).
    if direction not in ["1", "2"]:
        print("Invalid input! Defaulting to counter-clockwise.")
        direction = "2"
    vel_msg = Twist()
    #Specifies the forward linear velocity (1 meter per second).
    vel_msg.linear.x = 1.0
    # Angular velocity = v / r = 1 / radius (law of circular motion).
    vel_msg.angular.z = -1.0 / radius if direction == "1" else 1.0 / radius
    # We calculate the time for a complete revolution using the relationship:
    # Circumference = 2πr
    # Time = Distance / Linear Velocity
    # Here, linear velocity = 1 → time = 2πr / 1 = 2πr
    time = 2 * math.pi * radius / vel_msg.linear.x
    #Takes the present time as a starting point.
    t0 = rospy.Time.now().to_sec()
    rate = rospy.Rate(50)
    #The loop continues as long as the elapsed time is less than the time required to draw the circle.
    while rospy.Time.now().to_sec() - t0 < time:
        rospy.loginfo(f'x={pose.x:.2f}, y={pose.y:.2f}, radius={radius:.2f}')
        # Checks that the turtle doesn't go outside the window.
        # If it does go outside, it prints a message and stops.
        if not is_within_bounds(pose.x, pose.y):
            rospy.logwarn("Robot stopped! reached boundary of the turtlesim window.")
            break
        velocity_publisher.publish(vel_msg)
        rate.sleep()
    velocity_publisher.publish(Twist())
```

This function makes the turtle draw a full circle based on a user-specified radius and direction. It first validates the radius input, restricting it to a safe range ($0 < r \leq 4.0$), and asks whether the circle should be drawn clockwise or counterclockwise. Using the circular motion formula (angular velocity = linear velocity / radius), it sets the correct velocities. The total time to complete the circle is calculated using the circumference ($2\pi r$), assuming a constant speed of 1 m/s. A loop keeps publishing velocity commands until the turtle completes the circle or hits the boundary.

Draw spiral

```python
def draw_spiral(velocity_publisher):

    # The user is asked to enter the starting radius of the spiral.
    # If it's too small (< 0.1), it's set to 0.1.
    # If it's too large (> 5.0), it's reduced to 5.0 to stay within the visible window.
    start_radius = float(input("Enter starting radius: "))
    if start_radius < 0.1:
        rospy.logwarn("Too small! Using 0.1")
        return
    elif start_radius > 1.0:
        rospy.logwarn("Too large! Reducing to 1.0")
        return

    # The variable r represents the initial radius of the helical motion.
    # It starts at the user-defined value, controlling how tight the spiral begins.
    vel_msg = Twist()
    r = start_radius

    # Set the loop repetition rate to 10 times per second (10Hz).
    # This regulates the refresh rate to be smooth and regular.
    rate = rospy.Rate(10)

    # The loop continues as long as:
    # - The turtle is inside the turtlesim window.
    # - The radius r does not exceed 5.5 (so it doesn't go outside the window).
    while is_within_bounds(pose.x, pose.y) and r < 5.5:

        # Set the linear velocity so that it equals the current radius r.
        # This makes the turtle move faster as the circle gets larger.
        vel_msg.linear.x = r

        # Set the angular velocity to 1.0 (constant).
        # This causes the turtle to rotate at the same rate as the curve, but expand over time.
        vel_msg.angular.z = 1.0
        velocity_publisher.publish(vel_msg)
        rospy.loginfo(f'x={pose.x:.2f}, y={pose.y:.2f},increase in r={r:.2f}')
        # A very small increase in radius.
        # This gradual increase creates a spiral shape.
        # If it increases too much, the shape becomes unsmooth.
        r += 0.01
        rate.sleep()

    # After exiting the loop, send a zero-velocity message to stop the turtle.
    velocity_publisher.publish(Twist())

    # If the turtle exited the window, show a message.
    if not is_within_bounds(pose.x, pose.y):
        rospy.logwarn("Robot stopped: reached boundary of the turtlesim window")
```

This function makes the turtle draw a spiral by gradually increasing its linear speed while maintaining a constant angular velocity. It begins with a user-defined radius, adjusting it if the input is too small or large. Inside a loop, the turtle's radius increases slightly on each iteration, causing it to spiral outward. The motion continues as long as the turtle stays within the visible simulation window and the radius remains within limits. Once done, it stops the turtle and notifies the user if it exited the windo

Point to point motion

```python
def go_to_point(velocity_publisher):
    #The user is prompted to enter x and y values between 1 and 10.
    #The entered string is converted to a decimal for subsequent calculations.
    x_goal = float(input("Target x (1-10): "))
    y_goal = float(input("Target y (1-10): "))
    #A function that helps ensure that the target is inside the TurtleSim window (approximately 11x11 with margin).
    #If the target is outside the window, an error is logged via logerr and the function terminates.
    if not is_within_bounds(x_goal, y_goal):
        rospy.logerr("Coordinates out of range!")
        return
    #linear.x → forward speed
    #angular.z → yaw rate
    #Two proportional (Kp) gains:
    velocity_message = Twist()
    K_linear = 0.5
    K_angular = 4.0
    while True:
        # pose.x, pose.y are updated via the update_pose subscriber.
        # Threshold 0.1 m (≈ 10 cm) defines "arrival".
        distance = math.sqrt((x_goal - pose.x) ** 2 + (y_goal - pose.y) ** 2)
        if distance < 0.1:
            break
        #Linear speed grows with distance (slows automatically when close).
        #Desired heading from current pose to goal via atan2.
        #Heading error = desired - current yaw (pose.theta).
        #Angular speed proportional to that error.
        linear_speed = distance * K_linear
        desired_angle = math.atan2(y_goal - pose.y, x_goal - pose.x)
        angular_speed = (desired_angle - pose.theta) * K_angular
        #Sends the Twist message to /turtle1/cmd_vel.
        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed
        velocity_publisher.publish(velocity_message)
    #prints at most once every 0.3 s, even if the loop is faster.
    rospy.loginfo_throttle(
        0.3,
        f"pos=({pose.x:.2f},{pose.y:.2f})"
    )
    rospy.sleep(0.1)
    #Publishes a zero-velocity Twist, halting the turtle.
    velocity_message.linear.x = 0
    velocity_message.angular.z = 0
    velocity_publisher.publish(velocity_message)
    #Double-checks whether the turtle remained inside the window:
    #Outside → warning
    #Inside → success message with check-mark
    if not is_within_bounds(pose.x, pose.y):
        rospy.logwarn("Stopped: boundary reached.")
    else:
        rospy.loginfo("Target reached ✓")
```

This function moves the turtle from its current position to a user-specified point within the simulation window. It first checks that the destination coordinates are valid and within bounds. Using trigonometry, it calculates the angle needed to face the target and rotates the turtle accordingly. It then computes the distance to the target point and moves the turtle forward in a straight line. This allows accurate navigation to any point selected by the user.

Draw hexagon

```python
def draw_hexagon(velocity_publisher):
    length = float(input("enter the edge length of hexagon (<=2:)"))
    if length > 2:
       rospy.logwarn("Too large! reducing to 2")
       return

    for _ in range(6):
        # Move the turtle forward a distance equal to the length of the side
        move_linear(velocity_publisher, 1.0, length, True)
        #After each side, the turtle rotates 60 degrees.
        rotate(velocity_publisher, 30, 60, False)
```

This function commands the turtle to draw a regular hexagon by moving forward and turning 60 degrees after each side. It starts by asking the user for the edge length, with a maximum limit of 2 units to keep the drawing visible. It then loops six times to complete the hexagon, calling move_linear to go straight and rotate to make the angle turns. This creates a six-sided polygon with equal edges and angles.

Draw sinusoidal wave

```python
def draw_sine_wave(pub):
    """
    Let the user choose amplitude A, frequency f (cycles / m), and forward
    speed v.  The turtle is then driven along y = A·sin(2πf·s), where s is
    the arc-length already travelled.
    """

    # ---------- user input ----------
    A = float(input("Amplitude   A (0 < A ≤ 2.0 m): "))
    f = float(input("Frequency   f (0 < f ≤ 2.0 cycles/m): "))
    v = float(input("Forward speed v (0 < v ≤ 2.0 m/s): "))

    # ---------- basic validation ----------
    if not (0 < A <= 2.0 and 0 < f <= 2.0 and 0 < v <= 2.0):
        rospy.logwarn("Values out of range – using defaults A=1, f=0.5, v=1")
        return

    ω = 2.0 * math.pi * f          # angular frequency (rad / m)
    rate = rospy.Rate(60)              # 60 Hz for a smooth path
    twist = Twist()
    s = 0.0                            # distance travelled so far (m)

    while not rospy.is_shutdown():
        # slope dy/dx, then heading angle = atan(slope)
        slope            = A * ω * math.cos(ω * s)
        twist.linear.x    = v
        twist.angular.z  = math.atan(slope)

        if not is_within_bounds(pose.x, pose.y):
            rospy.logwarn("Boundary reached – stopping sine wave.")
            break

        pub.publish(twist)

        # advance arc-length: Δs = v · Δt   (Δt = 1 / 60 s)
        s += v * (1.0 / 60.0)
        rate.sleep()

    pub.publish(Twist())  # stop the turtle when finished
```

This function makes the turtle trace a sine wave pattern by combining forward motion with oscillating angular changes. The user inputs the amplitude, frequency, and forward speed, which shape the wave. Using the derivative of the sine function, the turtle's heading is adjusted with atan(amplitude × frequency × cos(frequency × t)) to follow the wave's slope. The turtle moves forward at a constant speed while continuously adjusting its direction based on the sine curve. The motion stops if the turtle exits the simulation window, ensuring it stays within bounds.

Reset method

```python
def reset_turtle(_unused=None):
    rospy.wait_for_service("/reset")
    try:
        reset = rospy.ServiceProxy('/reset', Empty)
        reset()
        rospy.loginfo("The turtle's location has been reset.")
    except rospy.ServiceException as e:
        rospy.logerr(f"Service call failed: {e}")
```

Main

```python
def main():
    rospy.init_node('turtle_motion_controller', anonymous=True)
    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
    rospy.Subscriber('/turtle1/pose', Pose, update_pose)

    rospy.sleep(1)

    while True:
        print("\nSelect one of the following motion trajectories for turtle robot:")
        print("0. Exit turtle\n1. Square\n2. Triangle\n3. Circular\n4. Spiral\n5. Point to Point\n6. hexagon\n7. Sine Wave\n8. Reset")
        choice = input("Enter your choice (0-8): ")
        if choice == '0':
            print("Exiting the program. Goodbye!")
            sys.exit()
        elif choice == '1':
            draw_square(velocity_publisher)
        elif choice == '2':
            draw_triangle(velocity_publisher)
        elif choice == '3':
            draw_circle(velocity_publisher)
        elif choice == '4':
            draw_spiral(velocity_publisher)
        elif choice == '5':
            go_to_point(velocity_publisher)
        elif choice == '6':
            draw_hexagon(velocity_publisher)
        elif choice == '7':
            draw_sine_wave(velocity_publisher)
        elif choice == '8':
            reset_turtle()
        else:
            print("Invalid choice!")

if __name__== '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
            pass
```

Choose from the menu to execute a motion pattern.

Menu Options

0. Exit turtle

1. Draw Square

2. Draw Triangle

3. Draw Circle

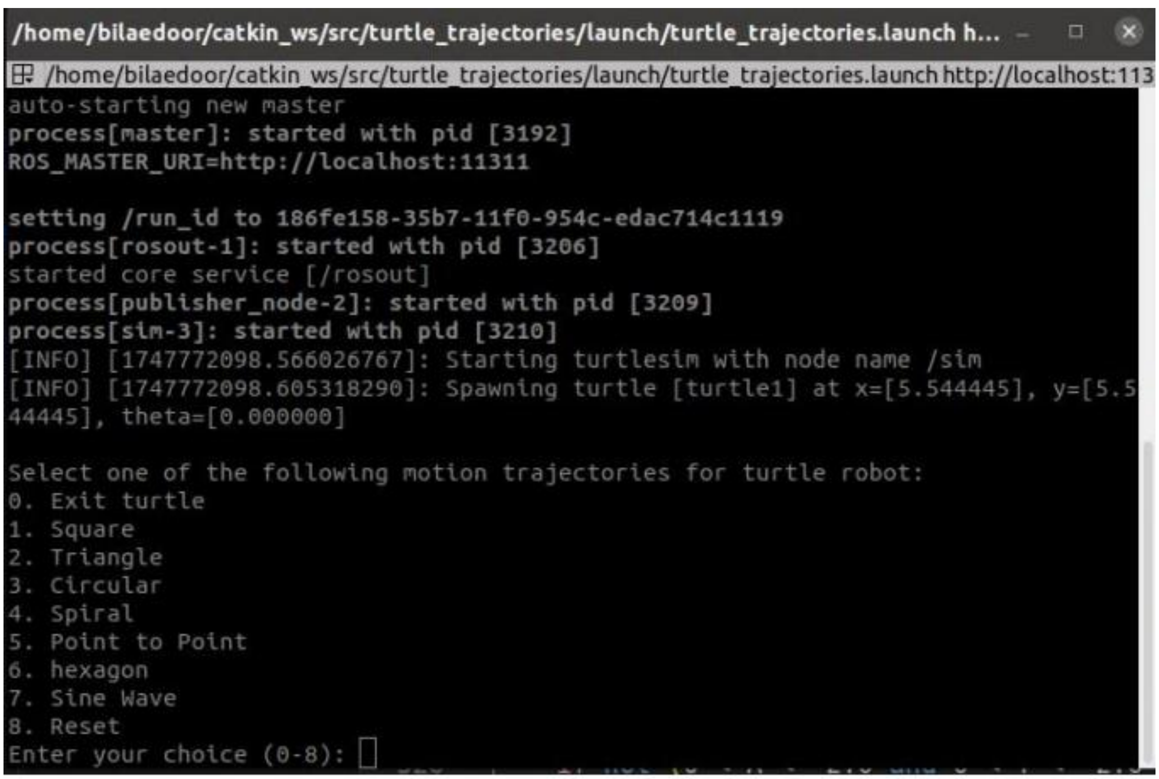
4. Draw Spiral

5. Move Point to Point

6. Draw Hexagon

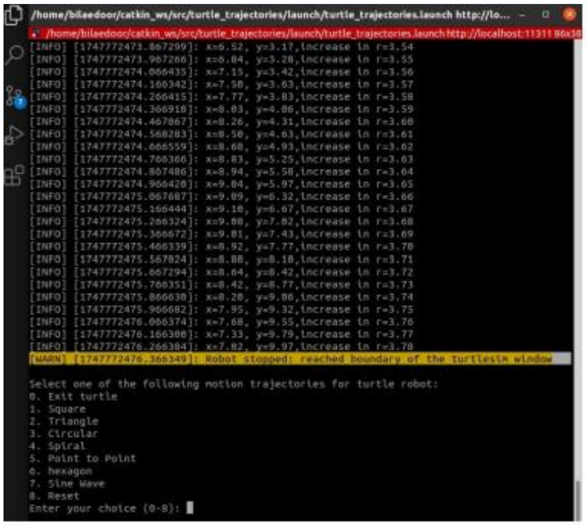
7. Draw Sine Wave

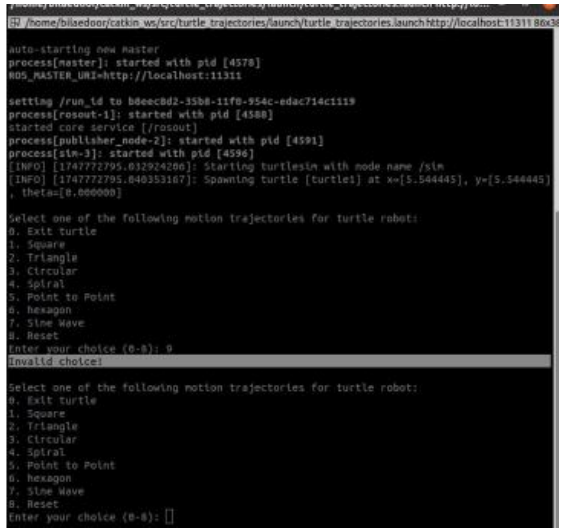
8. Reset Turtle



Safety & Constraints

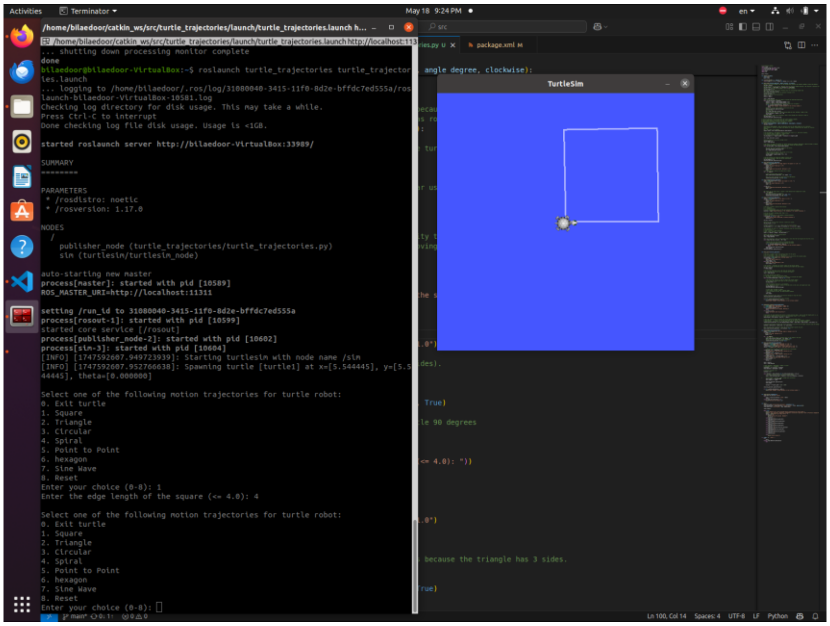
- Input validation is done to ensure the turtle remains within bounds.
- Movement stops automatically if the turtle is about to leave the window.
- Shapes are restricted to size limits suitable for the 11x11 Turtlesim window.



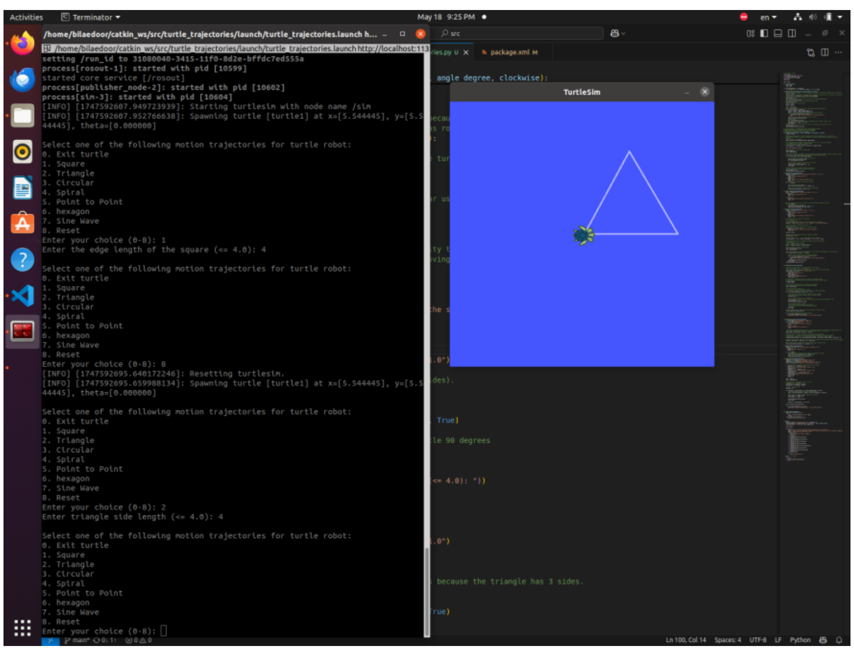


OUTPUT

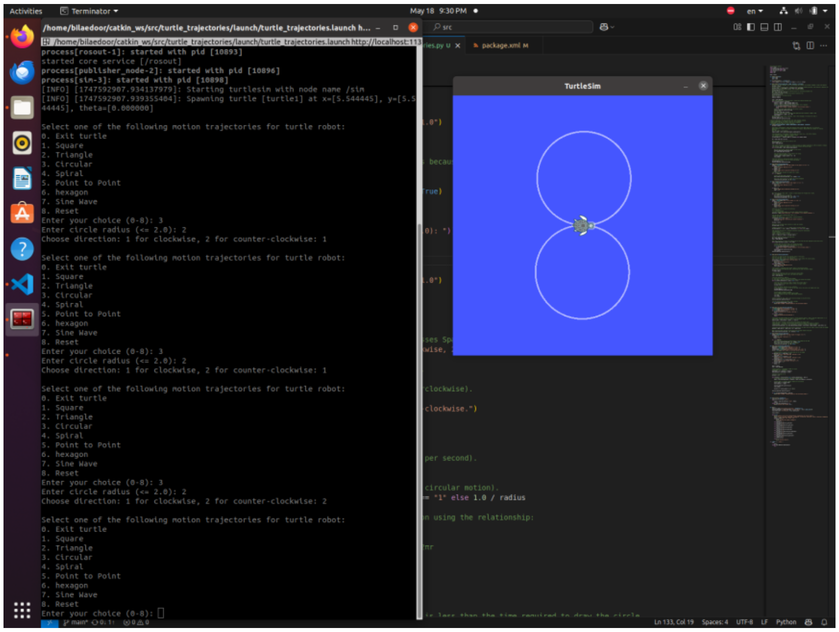
Square



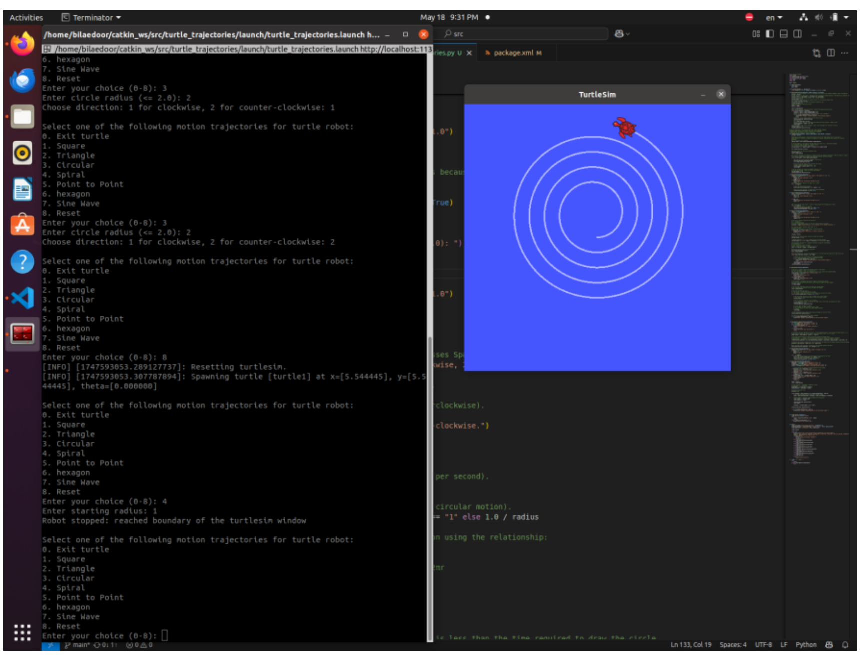
Tringle



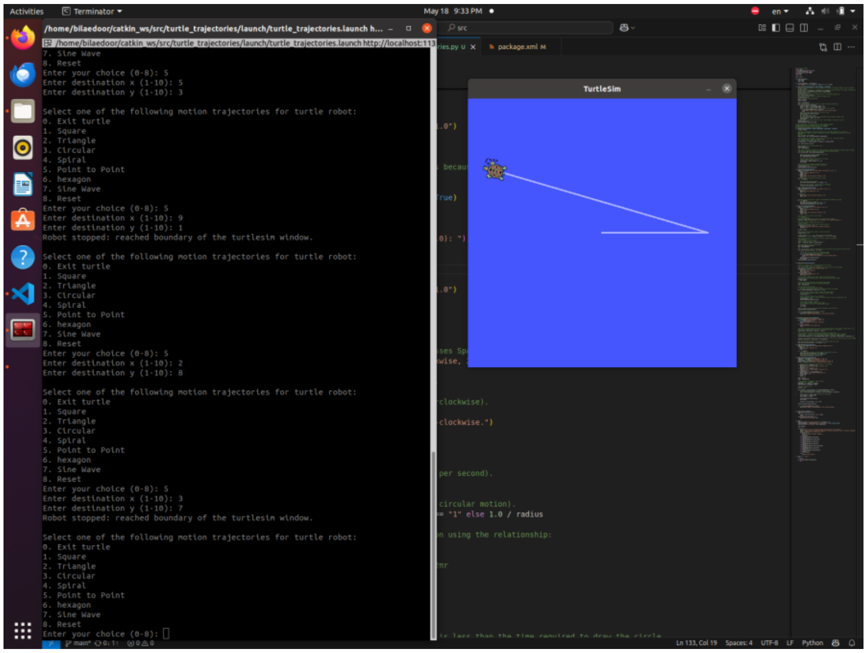
Circle



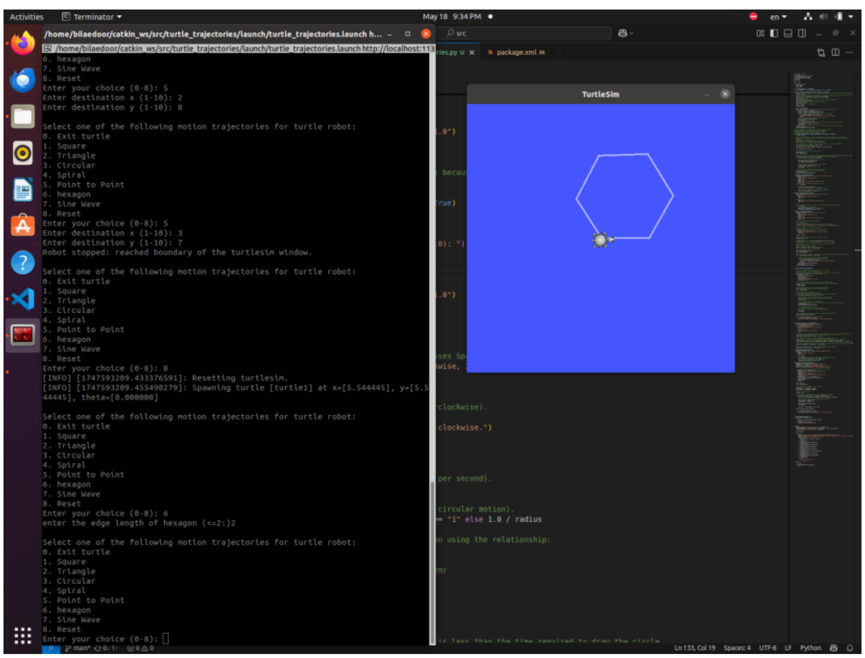
Spiral



Point to point



Hexagon



Sinusoidal wave