Edge Detection and Moving Along the Edge

By:

Bayat Vahabodin

Contents

Introduction

The following report describes two ROS2 nodes, namely Edge Detection and Move, created to detect edges and follow them. The Edge Detection node captures an input image from the camera, performs color thresholding, and uses Canny Edge detection to extract edges. It then publishes the edge image to the edges topic. The Move node subscribes to the edges topic, converts the edge image into a binary mask, extracts the largest contour, calculates its centroid, and generates a velocity command to follow the contour.

The goal of this project is to create two ROS2 nodes using Python that work together to detect edges from a camera feed and move a robot (Create3) along the detected edges. The first node is responsible for detecting the edges using the OpenCV library and publishing the detected edges to a topic named "edges". The second node subscribes to the "edges" topic, extracts the largest contour from the edges image, and calculates the error between the center of the contour and the center of the image. Based on this error, it publishes linear and angular velocity commands to move the robot along the edge.

Node Edge Detection

The node edge_detection is responsible for detecting edges in the input image and publishing the edges image as a ROS message. This node subscribes to the image topic and publishes the edges image on the edges topic.

The EdgeDetection class inherits from the Node class in the rclpy.node module. It creates a publisher on the edges topic with a queue size of 10 and a timer that calls the detect_edges method every 0.01 seconds. The CvBridge class from the cv_bridge module is used to convert the OpenCV image to a ROS message.

The detect_edges method captures frames from the video stream and converts them to the HSV color space. It then applies a color mask to isolate the red color and find the edges in the image using the Canny edge detection algorithm. Finally, it converts the edges image to a ROS message and publishes it on the edges topic.

Here's a step-by-step breakdown of what the method does:

1- Capture an image frame from the camera feed. The cap object is an instance of the cv2.VideoCapture class that is used to capture frames from the default camera.

```
ret, frame = cap.read()
```

2- Convert the image from the BGR color space to the HSV color space.

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

3- Define the range of red color in HSV. These values define the lower and upper bounds of the red color range in HSV.

```
lower_red = np.array([5,50,50])
upper_red = np.array([15,255,255])
```

4- Create a binary mask that threshold the HSV image based on the defined color range.

```
mask = cv2.inRange(hsv, lower_red, upper_red)
```

5- Apply the binary mask to the original image using the bitwise_and function. This step overlays the binary mask on the original image and keeps only the pixels that are within the defined color range.

```
res = cv2.bitwise_and(frame,frame, mask= mask)
```

6- Display the original image and the resulting edge map.

```
cv2.imshow('Original',frame)
cv2.imshow('Edges',edges)
```

7- Convert the resulting edge map to a ROS2 message using the CvBridge class and publishes it to the edges topic.

```
edges_msg = self.bridge.cv2_to_imgmsg(edges, encoding="passthrough")
edges_msg.header.stamp = self.get_clock().now().to_msg()
self.publisher_.publish(edges_msg)
```

Node Move

The node move is responsible for receiving the edges image and following the largest contour in the image. This node subscribes to the edges topic and publishes velocity commands on the cmd_vel topic.

The Move class inherits from the Node class in the rclpy.node module. It creates a publisher on the cmd_vel topic with a queue size of 10 and a subscription on the edges topic that calls the follow_edge method every time an image is received. The CvBridge class from the cv_bridge module is used to convert the ROS message to an OpenCV image.

The follow_edge method converts the edges image to a binary mask and finds the contours of the edges. It then chooses the largest contour as the edge to follow and calculates the centroid of the contour. If no contours are found or the area of the contour is zero, it stops moving. Otherwise, it calculates the error between the centroid and the center of the image and generates linear and angular velocity commands based on the error. Finally, it publishes the velocity commands on the cmd_vel topic.

Here's a step-by-step breakdown of what the method does:

1- Create a Move node that subscribes to the edges topic and publishes velocity commands to the cmd_vel topic.

```
class Move(Node):
    def __init__(self):
        super().__init__('move')
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        self.subscription_ = self.create_subscription(
            Image,
            'edges',
            self.follow_edge,
            10)
        self.bridge = CvBridge()
```

2- Define the follow_edge method, which is called each time a new edges image is received.

```
def follow_edge(self, msg):
   # Convert the ROS message to an OpenCV image
   edges = self.bridge.imgmsg_to_cv2(msg, desired_encoding="passthrough")

   # Convert the edges image to a binary mask
   thresh = cv2.threshold(edges, 128, 255, cv2.THRESH_BINARY)[1]

   # Find the contours of the edges
   contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

   # If no contours are found, stop moving
   if len(contours) == 0:
      twist = Twist()
      self.publisher_.publish(twist)
      return

   # Choose the largest contour as the edge to follow
   largest_contour = max(contours, key=cv2.contourArea)

   # Get the centroid of the largest contour
   moments = cv2.moments(largest_contour)
   # If the area of the contour is zero, stop moving
   if moments['m00'] == 0:
      twist = Twist()
      self.publisher_.publish(twist)
      return

   cx = int(moments['m10'] / moments['m00'])
   cy = int(moments['m01'] / moments['m00'])

   # Calculate the error between the centroid and the center of the image
   image_center_x = int(edges.shape[1] / 2)
   error = image_center_x - cx

   # Calculate the linear and angular velocity commands based on the error
   linear_vel = 0.2
   angular_vel = -0.002 * error

   # Publish the velocity commands
   twist = Twist()
   twist.linear.x = linear_vel
   twist.angular.z = angular_vel
   self.publisher_.publish(twist)
```

3- Initialize the node and start spinning

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

    node = Move()

    rclpy.spin(node)

    node.destroy_node()
    rclpy.shutdown()

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

Conclusion

In conclusion, these two nodes work together to allow the robot to detect edges and move along them. The edge_detection node detects edges in the input image and publishes the edges image, and the move node receives the edges image and generates velocity commands to follow the largest contour in the image. The robot can use these velocity commands to move along the detected edges.

To implement this system, we used the ROS2 framework and the OpenCV library for image processing. The edge_detection node subscribes to the robot's camera feed and uses the Canny edge detection algorithm to detect edges in the image. It then publishes the resulting edges image as a ROS2 message.

The move node subscribes to the edges image and processes it to find the largest contour, which represents the edge the robot should follow. It then generates velocity commands to move the robot along this contour.

Both nodes were implemented in Python and tested on an iRobot Create3 robot using ROS2 Humble. The code for these nodes can be found in this GitHub repository (https://github.com/vahabbayat/Create3-iRobot.git).