Appendix-C: Coding Standards/Convention

1. Overview

1.1 Purpose

The purpose of this document is to establish a set of coding standards and conventions to be followed by all team members working on the autonomous vehicle navigation project. Adhering to these guidelines ensures that our codebase remains clean, consistent, and maintainable.

1.2 Scope

These standards apply to all code written for the autonomous vehicle project, including Python scripts, ROS nodes, and integration code using the CARLA-ROS bridge.

2. General Guidelines

2.1 Readability

- Write clear and understandable code.
- Use meaningful variable and function names.
- Avoid complex and nested structures when possible.

2.2 Maintainability

- Ensure the code is easy to update and extend.
- Follow the DRY (Don't Repeat Yourself) principle.
- Write modular and reusable code components.

2.3 Consistency

- Maintain a consistent coding style throughout the project.
- Use consistent naming conventions and code structures.

3. Naming Conventions

3.1 Variables

- Use **snake_case** for variable names.

Example:

vehicle_speed = 0.0

obstacle_detected = False

3.2 Functions and Methods

- Use **snake_case** for function and method names.

```
def calculate_distance(point1, point2):
    pass
```

3.3 Classes

- Use **PascalCase** for class names.

```
class AutonomousVehicle:
    pass
```

3.4 Constants

- Use **UPPER_CASE** for constants.

```
MAX_SPEED = 30.0
MIN_SAFE_DISTANCE = 5.0
```

4. Commenting and Documentation

4.1 Inline Comments

- Use inline comments to explain complex logic or important steps.

```
# Calculate the distance between two points
distance = calculate_distance(point1, point2)
```

4.2 Docstrings

- Use triple-quoted docstrings for documenting modules, classes, and functions.

```
def calculate_distance(point1, point2):
"""
Calculate the Euclidean distance between two points.

Args:
    point1 (tuple): The (x, y) coordinates of the first point.
    point2 (tuple): The (x, y) coordinates of the second point.
Returns:
    float: The distance between the two points.
```

"""

pass

4.3 External Documentation

- Maintain external documentation to provide an overview of the project.

5. Code Structure and Organization

5.1 File Organization

- Organize code into meaningful directories (e.g., **scripts**).

5.2 Module Structure

- Ensure each module has a clear purpose and is self-contained.

5.3 Indentation and Spacing

- Use 4 spaces for indentation.
- Follow PEP 8 guidelines for spacing around operators and keywords.

6. Error Handling and Logging

6.1 Exception Handling

- Use try-except blocks to handle exceptions gracefully.

```
try:
    result = some_operation()
except Exception as e:
   rospy.logerr(f"Operation failed: {e}")
```

6.2 Logging Practices

- Use the **logging** module or ROS logging methods for logging.

```
rospy.loginfo("Starting the navigation module")
rospy.logwarn("Obstacle detected ahead")
rospy.logerr("Failed to connect to CARLA server")
```

7. Version Control

7.1 Commit Messages

- Write clear and concise commit messages.

[Feature] Add obstacle detection module

[Fix] Correct path planning algorithm bug

[Docs] Update artifacts with instructions

8. Testing and Quality Assurance

8.1 Code Reviews

- Conduct code reviews using tools like GitHub pull requests to ensure code quality.