Autonomous Drift Control

Code Launch Documentation

1 Connection

Raspberry Pi and Arduino Uno are initially installed on the machine. They are connected to a USB-B → USB-A cable. This connection provides the ability to read data from Arduino Uno via the Raspberry Pi using a serial port. Basically, the link to Arduino in Raspberry Pi OS (UNIX based OS) is /dev/tty* (for example, /dev/ttyACM0). After that, we can access the serial port of Arduino by using the Python library pyserial.

There an example of data reading by Raspberry Pi on Python.

```
import serial

SERIAL_PORT = "/dev/ttyACM0"
BAUD_RATE = 115200
port = serial.Serial(SERIAL_PORT, BOUD_RATE)
try:
    while True:
        line = ser.readline().decode('utf-8').strip()
        if line:
            data = line.split(',')
            # use data
except KeyboardInterrupt:
    ser.close()
```

To launch a code that runs automatic drift control and recovery from the slide, firstly, it is necessary to check the output format of Arduino Uno. For our car, the output looks like this:

$$\text{time}, a_x, a_y, \dot{\theta}, \text{Steer Angle}, \text{Gas} -$$

where a_x, a_y are the acceleration parts by the x and y axes accordingly, $\dot{\theta}-$ yaw rate of the car.

2 Main control code

The code mpc_new.py has following functional:

1. Neural Network (DynNet):
   - Purpose: Predicts the next-state delta (4D) given the current state (6D) and action (2D).
   - Model Architecture: 2 hidden layers with 128 ReLU-activated neurons each.
   - File: The trained model is loaded from dyn_v3.pt.
2. Constants:
   - Define RC car hardware limits (steering/gas PWM ranges), target yaw rate, PID gains, MPC hyperparameters, etc.
   - These values can be tuned for different car setups.
3. Model Loading:
   - Loads the trained dynamics model from dyn_v3.pt.
   - Normalization statistics (mu, sigma) are extracted from the checkpoint for proper input/output scaling.
4. Normalization and MPC:

- Normalization functions (_norm, _denorm) ensure the model works with standardized inputs/outputs.
- MPC controller (mpc_control):
  - Samples random action sequences.
  - Predicts state evolution using the neural network.
  - Selects the first action of the sequence with the lowest cost (balancing yaw, lateral acceleration, and slip).

5. Main Control Loop (FSM):

   This is where the real-time control happens. It includes:

   - Serial Setup and Arduino Reset
     - Opens serial connection with Arduino.
     - Resets Arduino automatically (reset_arduino()).
     - Waits until valid IMU data is received before starting.
   - Finite State Machine (FSM)

     The car operates in three states:
     - DRIFT (default start): Uses a PID controller to maintain YAW_SP (target yaw rate). Increments lap count based on integrated yaw angle (angle_accum).
     - RECOVERY: Triggered after completing TARGET_LAPS. Runs mpc_control() to stabilize the car. Transitions to IDLE once yaw rate and lateral acceleration drop below thresholds.
     - IDLE: Keeps the car stationary (steer_cmd = STEER_C, GAS_CMD = GAS_MIN).
   - Command Sending:

     At each loop iteration, the chosen steer_cmd and GAS_CMD are sent to Arduino as:

     $$\text{"}\{\text{steer\_pwm}\},\{\text{gas\_pwm}\}n\text{n"}$$
   - Error Handling:

     Skips corrupted or incomplete serial lines (try/except around parsing).

3 Code launch

Firstly, in directory .../Autonomous-drift-control/ML/ should be set up a python virtual environment by command:

```
python -m venv venv
```

Then, after venv creation, it can be activated by this command:

```
source venv/bin/activate
```

Therefore, in venv libraries must be installed by command:

```
pip install -r ../requirements.txt
```

After all steps, control code could be launched by command:

```
python mpc_new.py
```

Note, that serial port can be not the same as in current code.