**How to Use CARLA**

**Windows**

1. Run anaconda3 as admin
2. Type conda env export --no-builds > env\_carla99.yml
3. Type conda env create -f env\_carla99.yml -n carla99

Note: version 9.12 is compatible with Python 3.7 and 3.8. Need to specify in conda environment

1. Type conda activate carla99
2. Add the following to PYTHONPATH in the system Environment Variables:
   1. D:\Windows\CARLA\CARLA\_0.9.12\PythonAPI\carla\dist\carla-0.9.12-py3.7-win-amd64.egg
   2. D:\Windows\CARLA\CARLA\_0.9.12\PythonAPI\carla
   3. D:\Windows\CARLA\CARLA\_0.9.12\PythonAPI

To run:

1. Start the server by running CarlaUE4.exe
2. To change Town: navigate to ~PythonAPI\util and type python config.py -m Town04
3. Navigate to ~carla-takeover-client-main and type python spawn\_locations.py
4. Type python synchronous\_client.py -sp test5.json -sc scenario\_configs/bike.json -r True
5. Type python npc\_manager.py -sp test5.json

**Linux**

1. Run anaconda3 as admin
2. Type conda env export --no-builds > env\_carla99.yml
3. Type conda env create -f env\_carla99.yml -n carla99

To run:

1. Type conda activate carla99
2. Type export PYTHONPATH=/media/homlabadmin/CE3A4C113A4BF4CD/Ubuntu/CARLA/CARLA\_0.9.12/PythonAPI/carla

Launch CARLA server first (CarlaUE4.sh) and then run launch\_client.py. Server needs to run before the client

* Don't need the 'record', otherwise it will record every time

Note:

* Agent files (basic\_agent.py, behavior\_agent.py, behavior\_types.py, controller.py) in the core folder are from the source files. They are modified to increase the speed and make the agent go to the destination.
* Npc speed can be changed in the npc\_manager.py line 13. Higher value indicates *slower* speed

**To use a different Town (Linux):**

* Note: recorder-client-master folder needs to be outside (not in CARLA folder). setup\_server.sh is going to look for the CARLA folder in a specific way, needs to be outside for it to work
* Close CARLA server

1. Need to change path in setup\_server.sh: go to “carlaPath” line3 → (open a Terminal → type pwd) → copy the path of the CARLA folder e.g. "/media/homlabadmin/CE3A4C113A4BF4CD/Ubuntu/CARLA/CARLA\_0.9.12" → Save → in Terminal, type ./setup\_server.sh -m Town04 to launch the server
2. The configuration file and scenario are designed for Town03, but you can generate new scenarios for Town04
   * Have to change spawn points because spawn points are for Town03: server should be open in Town04 → green/red dots show where the npc will be spawning → type python spawn\_locations.py → open launch\_client.sh → go to spawn\_config line → when you ran spawn\_locations.py you generated a new test3.json. Need to change path to that file by deleting “spawn\_configs” → Open Terminal, type ./setup\_server.sh -m Town04 → type ./launch\_client.sh -npc →
   * To close npc: press enter in the Terminal
   * Note: Bike crossing scenario would not work here because it was created for Town03

**To close client:** close the Terminal

**To change scenarios:**

1. Open scenario\_configs folder → open bike.json - configurations for scenario → "dist" 1 and 2 are trigger distances. If 150m away, it will start giving the alarm. If 50m away, it will take you off Autopilot and brake the car.
2. Can change values BUT they have to be strings with double curly braces around them. Can't type in integer or float. Won't be able to save
   * "bikes": spawn point for the bike in Town03
   * Can change the actor\_type e.g. can make a pedestrian cross
   * If want the bike to move faster, change the "throttle" value
   * Make a copy of the file and change values if you want new scenarios
   * Bike\_crossing.py: runs the bike scenario - don't need to make any changes here. Changes only have to be made to the configuration file

**Logitech steering wheel:** file is wheel\_config.ini

* If want to add your own controller: copy the code → add a new section just like the two above, called "G27 Racing Wheel" → do your own key mappings - assign actions to buttons you want
* Note: it looks for a controller first, so if you want to use keyboard you'll have to unplug the controller

**To change the number of npcs surrounding my vehicle:**

1. Make sure server is running and current pygame is closed → type python spawn\_locations.py → will see spawn locations in the server window - where npcs will spawn. If not happy with it, run the same command again to get new spawn locations. It randomly generates new spawn locations.
2. Flags to change number of npcs spawned: type python spawn\_locations.py -cr 30 → type ./launch\_client.sh -npc
   * -cr: radius - default is 70. if you make it smaller, cars will be more concentrated
   * -cn: number of npc vehicles spawned in the map. More vehicles will make it slower
   * For list of flags: open spawn\_locations.py → go to spawn\_info: list of tuples generated by pair\_sp\_with\_bp function line 177
   * Note: when using Town06, run spawn\_location.py without pedestrians, to avoid cars colliding with pedestrians that walk on the street, you can do this with the flag python spawn\_locations.py --ped-num 0

**To change the warning message:**

1. Go to assets folder → icons / sounds
2. To change message text: go to synchronous\_client.py → line 377

**To change pygame windows size:** go to synchronous\_client.py → lines 47 and 48

**How to check if it is continuously surrounded by npcs**:

1. Check the Saved recordings: Saved folder → open the saved recordings → targets → each .json file is a timestep. If you open them, you will see the list of pedestrians. Each timestep is a json object, within each object there are keys and values. One of the values would be list of pedestrians, npc cars
   * "car\_vx/vy" - your own car
   * "type\_id" - any npc (pedestrians and vehicles). Separate lists for pedestrians and cars
2. Easier way to view data: open browser → <https://jsongrid.com/json-grid> → copy/paste contents of the file. This is for quick view of the file.
   * If you need to analyse the entire data, use a python script to parse through all of it. Use Jupyter notebook
   * Good practice to not combine them because every timestep is one snapshot of the world, so don't want to combine different timesteps

Note: .json files are every time step. It's in 15Hz so 1/15 of a second

**How to check when Autopilot was engaged:**

1. In the Grid view → stage → 0 means neutral, nothing happening, just driving along. Every time the number changes it goes into a different stage. Autopilot is stage 3 or 4 - when stage shows 3 or 4, autopilot disengages
   * With the python script to plot, you can see exactly where it happened. Will not need to go into each file to check
2. Need to close the client for it to stop recording

**Code to disengage Autopilot (already added):**

Lines 157, Lines 218-269

**Note**

print\_spectator\_coord.py prints the spectator’s current position as x, y, z coordinates to a terminal when you ‘fly’ around in the server map. Can be used to set locations for the client.

**Data Output:** all shown in Grid

* Gametime = simulation clock time
* Global in the games coordinate system, how far are you from the origin of the world
* Local - according to the car's coordinate system
* cte = cross track error, largest error is about 0.2m
* Simulator uses the line of the lane you are closest to as the 0 value
  + If the car shifts its lane, it will recalculate the value and start tracking that lane instead
  + To identify the moment of lane shift from lane to another: If you plot the cte, you will see a sudden jump. First need to convert values to a table
* k: usually very small cause it is 1/turning radius

*Variable names:*

* gametime: time passed in the simulated world
* car\_vx, car\_vy, ... : velocities in the vehicle reference frame
* global\_...: motion states in the global frame (referenced from some fixed origin in the world)
* stage: variable used by the state machine for showing warning and disengaging autopilot. for what happens in each stage, please look at the code in synchronous\_client.py
  + 1 = flash warning
  + 2 = bike starts crossing
  + 3 = autopilot toggled off
* headingerror: angular difference between lane and car
* cte: lateral difference between lane and car
* k: road curvature
* dist\_to\_car: distance to nearest car in front. if there's no car, defaults to 50
* dist\_to\_walker: distance to nearest pedestrian in front, also defaults to 50
* is\_junction: boolean on whether the car is at a junction
* land\_id: which lane the car is in