AirSim Project

Parinaz Akef

Spring 2025

Introduction

The objective of this project is to deepen understanding of the AirSim library, utilize it to collect data from a drone, and ultimately employ reinforcement learning algorithms for training purposes in future work.

Github repo link for codes:

https://github.com/Parinaz11/AirSim-Drone-Project.git

AirSim and Unreal Engine 4.27 Installation Procedure

Create an account on the Epic Games Store. This account is required to download the Epic Games Launcher

Unreal 4.27

Install Epic Games Launcher

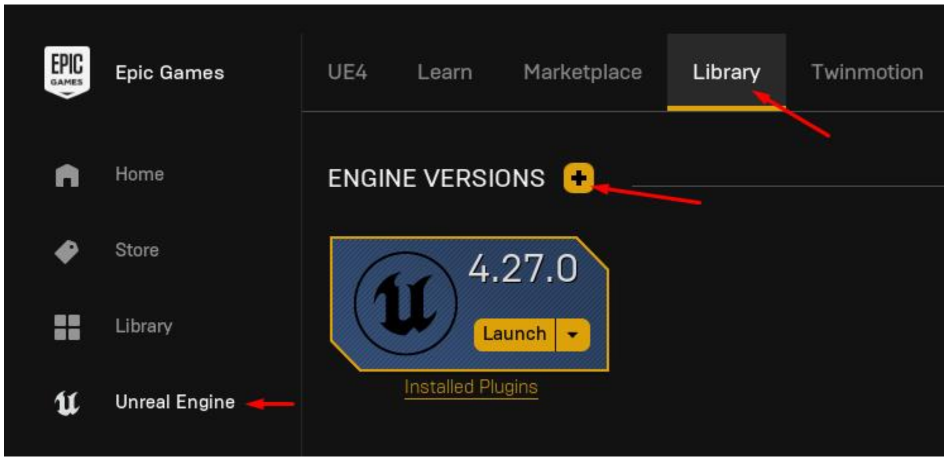
- Navigate to the Epic Games Store website
- Download and install the Epic Games Launcher
- Log in using the created account

Network Preparation

For network preparation on a Windows 11 system targeting access to restricted resources, I employed 403 online (Shekan) DNS service as a strategic solution to bypass regional network constraints. By reconfiguring DNS settings, I had access to Epic Games resources.

When I opened the Epic Games Launcher and tried to download Unreal Engine 4.27 by clicking the "+" button, the download consistently failed. Initially, I was frustrated because no matter how many times I clicked "Download", nothing happened. After troubleshooting, I discovered the problem wasn't just regional restrictions, but my Windows Firewall blocking the application. By accessing Windows settings and specifically allowing the Epic

Games Launcher through the firewall, I resolved the connectivity issue and enabled the download to proceed successfully. (almost 14GB)



AirSim Clone

For cloning the AirSim repository:

git clone https://github.com/microsoft/AirSim.git

During the AirSim repository setup, I encountered unexpected challenges with the cloning process, which repeatedly failed midway through. To circumvent this issue, I adopted two alternative approaches: first, I directly downloaded the repository as a ZIP file (AirSim-main.zip) from GitHub, which resolved the initial cloning problems. Subsequently, I discovered that cloning the repository through Visual Studio also proved successful, providing a reliable alternative method for obtaining the project files.

Visual Studio 2022 Setup for AirSim

Not sure if everything here is *strictly* necessary, but this config built things without errors.

1. Install Visual Studio 2022

- **Use the Visual Studio Installer** (don't grab VS 2019—AirSim might throw errors with it).
- **Workloads to check:**
  - ☑ **Desktop development with C++** (mandatory)
  - ☑ **Python development** (not sure if needed, but I had it checked)

    - ☑ **Data science and analytical applications** (definitely not needed, just in case)

2. Individual Components

Under the **Individual components** tab, make sure these are selected:

- ☑ **.NET Framework 4.8.1 SDK**
- ☑ **Windows 10 SDK (10.0.19041.0)** → *AirSim might not play nice with other SDK versions.*

3. Under "Desktop development with C++" (right-side panel)

- Both of these were checked for me:
    - ☑ **Windows 11 SDK (10.0.22621.0)**
    - ☑ **Windows 10 SDK (10.0.19041.0)**
    - *No clue if both are needed, but it worked, so... ¯_(ツ)_/¯*

Final Step

Hit **Install** and let it do its thing.

AirSim Build

Now it's time to actually take build of AirSim.

You should add eigen library manually through the eigen-3.3.7.zip file. Just put the contents in directory \AirLib\deps\eigen3. Delete or rename the bench folder in this directory (if you don't, it would later result in error for the final run)

Open visual studio 2022, select

$$continue\ without\ code\ \rightarrow\ Tools\ \rightarrow\ Command\ Line\ \rightarrow\ Developer\ Command\ Prompt$$

Nagivate to AirSim folder with build.cmd file

Write in cmd:

Set UE4_ROOT="D:\Epic_Games\v4\UE_4.2"

→ This is the path that I installed Unreal 4.27 on, and it helps AirSim to find Unreal Engine.

Now run the command **build.cmd** and when the build is successful you're done.

Running AirSim on Unreal Engine 4.27 – Setup Guide

Please follow each step carefully and in the exact order to ensure a successful build and launch.

1. Create a New Unreal Project

- Launch **Unreal Engine 4.27**.
- Select **Games** from the project categories.
- Choose the **Blueprint** project template.
- Click **Next**, and then create your project with a suitable name and location. This will generate an empty Blueprint-based project.

2. Add a C++ Class (Required for Successful Execution)

- After the project has been created, open it in the Unreal Engine editor.
- Navigate to **File** → **New C++ Class**. You can select any class (e.g., None or Actor), as the purpose here is only to ensure Unreal generates the necessary project files for C++ integration.
- Click **Create Class** and allow the engine to compile the necessary files. This step is essential for compatibility with AirSim.

3. Add the AirSim Plugin

- Locate your **AirSim** installation directory and navigate to: AirSim/Unreal/
- Copy the folder named **"Plugins"**.
- Paste this **Plugins** folder into the root of your newly created Unreal project directory, **next to the .uproject file**.

4. Modify the .uproject File

- Open the .uproject file using **VS Code**, **Notepad++**, or any other text/code editor.
- Add the following code snippet inside the Modules or Plugins array (depending on where plugin definitions are in your .uproject structure):

  AirSimTestV0 → Is the name of my Unreal project

```
{
            "FileVersion": 3,
            "EngineAssociation": "4.27",
            "Category": "",
            "Description": "",
            "Modules": [
                        {
                                    "Name": "AirSimTestV0",
                                    "Type": "Runtime",
                                    "LoadingPhase": "Default",
                                    "AdditionalDependencies": [
          "AirSim"
        ]
                        }
            ],
            "Plugins": [
                        {
                                    "Name": "AirSim",
                                    "Enabled": true
                        }
   ]
}
```

After integrating the AirSim plugin and updating the .uproject file, proceed with the following steps to prepare the project for development and execution:

1. **Generate Visual Studio Project Files**
   - Right-click on your Unreal project's .uproject file.
   - Select **"Generate Visual Studio project files"** from the context menu.
   - This will create a .sln (solution) file and configure your C++ project for Visual Studio.
2. **Open the Solution in Visual Studio**
   - Navigate to your project directory.
   - Double-click on the newly generated .sln file to open it in **Visual Studio**.
3. **Build and Run the Project**
   - Ensure that **Developer Mode** is enabled in your system settings. This is required for full functionality.
   - With the solution open in Visual Studio, press **F5** to build and launch the project.

- Visual Studio will compile the necessary files and open the Unreal Engine editor once successful.

Helper Links:

- https://github.com/1508189250/AirSim/blob/master/docs/build.md (Good)
- https://github.com/microsoft/AirSim/blob/main/docs/build_windows.md

Getting Familiar with settings.json

In settings.json file which is in the Documents/AirSim/settings.json:

Drone only config & Car Config:

```
{
        "SettingsVersion": 1.2,
        "SimMode": "Multirotor",
        "Vehicles": {
                "Drone1": {
                "VehicleType": "SimpleFlight",
                "X": 0, "Y": 0, "Z": -5
                }
        }
}
```

```
{
        "SettingsVersion": 1.2,
        "SimMode": "Car",
        "Vehicles": {
                "Car1": {
                "VehicleType": "PhysXCar”,
                "X": 0, "Y": 0, "Z": -5
                }
        }
}
```

The settings.json file is in JSON format and should be saved in ASCII to avoid issues. Below are the primary non-sensor settings you can modify to customize the drone and environment, along with their impact and testing steps.

Simulation Mode (SimMode)

We can switch between simulation modes:

Multirotor Mode:

- Set "SimMode": "Multirotor" in settings.json.
- Run test_straight_line to confirm the drone follows

ComputerVision Mode:

- Set "SimMode": "ComputerVision".
- Run the same path functions and observe if physics is disabled (drone moves precisely without flight dynamics).

Note: I tested and Unreal engine crashed, and because physics here are not abled and we want to use the drone with physics, I did not proceed any further.

Vehicle Configuration (Vehicles)

We want to modify drone type, initial position/orientation, and control settings

For Single Drone Setup:

VehicleType: Use "VehicleType": "SimpleFlight" (default) for straightforward control without external flight controllers, which is compatible with our path-following scripts. Alternatively, we can use "VehicleType": "PX4Multirotor" for more realistic PX4-based control, but this requires additional setup (e.g., TCP/UDP ports).

Initial Position and Orientation: Next, we set `X`, `Y`, `Z` (in NED coordinates, meters) and `Yaw`, `Pitch`, `Roll` (in degrees) to position the drone at the start of our path.

DefaultVehicleState: Next, we set "DefaultVehicleState": "Inactive" to prevent propellers from spinning until armed via API, which useful for testing path initialization.

AllowAPIAlways: Set "AllowAPIAlways": true to ensure API control without requiring a remote controller.

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",
  "Vehicles": {
          "Drone1": {
                  "VehicleType": "SimpleFlight",
                  "X": 0, "Y": 0, "Z": 0,
                  "Yaw": 0, "Pitch": 0, "Roll": 0,
                  "DefaultVehicleState": "Inactive",
                  "AllowAPIAlways": true
   }
}
```

Code: In file "test_straight_line.py"

(The code for part 1, getting familiar with settings.json file)

For Multiple Drones Setup:

Now we have two drones, and when we run the test_straight_line.py script, only one of them moves. The settings.json file:

```
{
        "SettingsVersion": 1.2,
        "SimMode": "Multirotor",
        "Vehicles": {
                "Drone1": {
                        "VehicleType": "SimpleFlight",
                        "X": 0, "Y": 0, "Z": 0,
                        "Yaw": 0
                },
                "Drone2": {
                        "VehicleType": "SimpleFlight",
                        "X": 1, "Y": 0, "Z": 0,
                        "Yaw": 90
                }
        }
}
```

Clock Speed (ClockSpeed)

For controling the simulation’s speed relative to real time, affecting how quickly the drone completes paths.

Modification:

- Set "ClockSpeed": 1.0 for real-time simulation (default).
- Use "ClockSpeed": 5.0 to speed up simulation (5x faster, e.g., our 6-second path completes in almost 1.2 seconds).
- Use "ClockSpeed": 0.1 to slow down simulation (10x slower, e.g., 60 seconds for the path) for detailed debugging.

For testing, I changed the clockSpeed to 5.0 (to be 5x faster) and it worked faster:

```
{
        "SettingsVersion": 1.2,
        "SimMode": "Multirotor",
        "ClockSpeed": 5.0,
        "Vehicles": {
                "Drone1": {
                        "VehicleType": "SimpleFlight",
                        "X": 0, "Y": 0, "Z": 0
                }
        }
}
```

Recording Settings (Recording)

For Logging drone data (position, orientation, velocity) during path execution, which is useful for analyzing our straight, triangular or half-circle paths.

Modification:

- Enable recording with "Enabled": true to start logging automatically.
- Set "RecordInterval": 0.1 to log data every 0.1 seconds.
- Use "RecordOnMove": true to log only when the drone's position or orientation changes, reducing file size for static moments.
- Specify a custom folder with "Folder": "/path/to/recordings" for organized data storage.

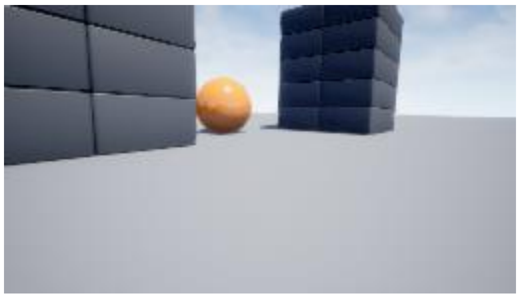
Testing: When we set the recording parameters, the output was a file named airsim_rec.txt with content:

```
VehicleName  TimeStamp       POS_X  POS_Y     POS_Z  Q_W  Q_X  Q_Y  Q_Z  ImageFile
Drone1       1748872557432 0        0         0.318197  1    -0   0    0    img_Drone1__0_1748872557226795500.png
Drone1       1748872557510 0        0         0.556437  1    -0   0    0    img_Drone1__0_1748872557252128500.png
Drone1       1748872557693 0        0         0.681403  1    0    0    0    img_Drone1__0_1748872557313270000.png
Drone1       1748872557870 0        0         0.681403  1    0    0    0    img_Drone1__0_1748872557373294100.png
```

And continuing to:

```
Drone1    1748872580529 3.01182e-06   3.5128e-06    -1.97141    0.999988    0.00136569    -0.00123506    0.00449085    img_Drone1__0_1748872565166441300.png
Drone1    1748872580664 0.000787727   0.000838163   -2.24848    0.998126    0.0146013     -0.0157183     0.0573041     img_Drone1__0_1748872565212094800.png
Drone1    1748872580805 0.00648695    0.00679285    -2.51335    0.985363    0.0320878     -0.0432176     0.16175       img_Drone1__0_1748872565259189200.png
Drone1    1748872580946 0.0205543     0.02139       -2.70103    0.953167    0.0401562     -0.07352       0.290612      img_Drone1__0_1748872565306800200.png
```

And next to this text file, a folder named “images” is showing the images that the drone recorded. Such as:



The settings.json file:

```
{
        "SettingsVersion": 1.2,
        "SimMode": "Multirotor",
        "Recording": {
                "Enabled": true,
                "RecordInterval": 0.1,
                "RecordOnMove": true,
                "Folder": "C:\\Users\\parin\\Documents\\MyRecordingsTest"
        },
        "Vehicles": {
                "Drone1": {
                "VehicleType": "SimpleFlight",
                "X": 0, "Y": 0, "Z": 0
                }
        }
}
```

Analyzing and plotting the results:

We can also plot the recorded positions using a Python script (e.g., with matplotlib and pandas) to visualize the path, with this python script and the CSV file that we got (airsim_rec.txt)

Environment Settings (OriginGeopoint and Wind)

Next, we modify geographic origin and wind to test path robustness under environmental conditions. (geographic origin and environmental conditions like wind, affecting path accuracy)

OriginGeopoint

Defines the latitude, longitude, and altitude of the Unreal environment's Player Start component, useful for realistic positioning.

Wind: Simulates wind in NED coordinates (m/s) to test path robustness (e.g., how our path handles crosswinds).

- Example: "Wind": [5, 0, 0] for a 5 m/s wind in the positive x-direction.
- OR, Set dynamically via API: client.simSetWind(airsim.Vector3r(5, 0, 0)).

Example in settings.json:

```
{
        "SettingsVersion": 1.2,
        "SimMode": "Multirotor",
        "OriginGeopoint": {
                "Latitude": 47.641468,
                "Longitude": -122.140165,
                "Altitude": 0
        },
        "Wind": [5, 0, 0],
        "Vehicles": {
                ... (same as before)
        }
}
```

Part 2: Connecting to AirSim Using Python API

For writing the python script we need to know how AirSim's coordinate system works for its APIs.

AirSim uses the **NED (North, East, Down)** coordinate system for its APIs, where:

- +X is North
- +Y is East
- +Z is Down (positive downward, unlike Unreal Engine's Z-up convention)

All units are in SI (meters for distance, radian for angles unless specified otherwise e.g., degrees in settings).

Note: This differs from **Unreal Engine's coordinate system** (which +X is forward, +Y is right and +Z is up, Z opposite to NED's down). Also, units are in centimeters. AirSim handles conversions internally (by doing a mapping, which understanding it is key for precise drone control in python)

Mapping Unreal Engine Coordinates to AirSim NED Coordinates

Position Conversion:

AirSim's NED origin (0, 0, 0) is set at the **Player Start** components in Unreal. It can be configured via the *OriginGeopoint* in *settings.json* for geographic coordinates such as Latitude (Definiton: Latitude measure how far north or south a point if from the Equator. Equator is an imaginary line that circles that Earth exactly halfway between the North Pole and the South Pole. It divides the Earth into the Northerm Hemisphere and the Southern Hemisphere and it's at 0 degrees latitude. The Equator is the reference line from which latitude is measured north or south. In simple terms, it's the Earth's "waistline" around its middle. Range: It ranges from 0° at the Equator to 90° North at the North Pole and the same 90° south at the South Pole. Example: A latitude of 40°N means the location is 40 degrees north of the Equator), Longitude (Definition: Longitude measures how far east or west a point is from the Prime Meridian. The Prime Meridian is the vertical imaginary line at 0° longitude, dividing the Earth into Eastern and Western Hemispheres. Range: It ranges from 0° at the Prime Meridian to 180° East or West. A longitude of 74°W means the location is 74 degrees west of the Prime Meridian), Altitude (Definition: Altitude is the height of a point relative to a reference level, usually mean sea level, meaning the average level of the ocean's surface, measured over a long period, taking into account the natural variations caused by tides, waves and weather. Units: Typically measured in meters or feet.)

Formula for converting Unreal coordinates (in cm) to AirSim NED (in meters):

- NED_X = (Unreal_X - Offset_X) 0.01
- NED_Y = (Unreal_Y - Offset_Y) 0.01
- NED_Z = -(Unreal_Z - Offset_Z) 0.01

Formula Explanation: Subtracting **Player Start** (Player Start is an object or marker that defines where the player character initially appears in the level) **offset** (Player Start offset is the vector difference between the Player Start's position and the world origin). Why subtract? So if you have a location relative to the Player Start, subtracting this offset will align that location to the environment's global coordinate system instead of relative to the Player Start. Scale by 0.01 (cm to meters). And finally, Invert the Z-axis (Unreal's +Z up becomes NED's -Z down)

Rotation Conversion:

Unreal uses a Z-up right-handed system (**Right-handed** means the coordinate axes (which are X axis, Y axis and X axis) follow the right-hand rule. If you point your right hand's thumb (X axis), index (Y axis), and middle finger (Z axis) so they are perpendicular), with **rotations in degrees** (meaning that for each axis (X, Y, and Z), the rotation is measured in degrees from its previous orientation), while AirSim uses NED (left-handed, because of being Z-down instead of Unreal's Z-up) with **rotations often in quaternions or Euler** angles (yaw, pitch, roll) in radians.

Euler Angles (Yaw, Pitch, Roll):

Represents rotation as three separate angles (the three mentioned above). Imagine rotating an object by turning first around one axis, then another, then another. It's easy for humans to understand but they suffer from gimbal lock (where you can lose a degree of freedom at certain rotations). And it's usually expressed in radians or degrees.

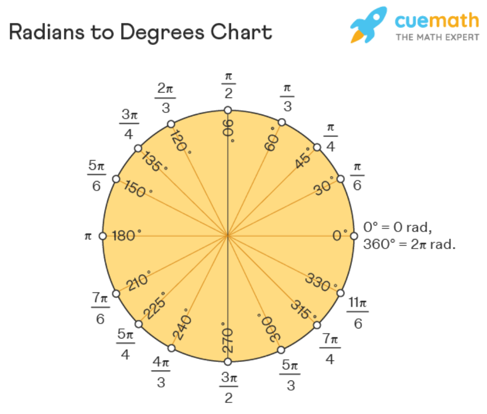


Figure 2 – Displaying Radian (the ones with pi) in a circle

Yaw: Rotation around Z-axis (down), +ve is clockwise from North (X-axis) (+ve (positive) means clockwise rotation around the Z-axis and -ve (negative) means counterclockwise rotation)

Pitch: Rotation around Y (East), +ve tilts nose down

Roll: Rotation around X (North), +ve rolls right

(<u>note</u>: I didn't quite understand the +ve)

Quaternions:

Uses four numbers (one real part and three imaginary parts) to represent rotation. It harder to read directly but avoids gimbal lock, making them better for smooth and stable rotations. It's efficient for computing rotations and blending multiple rotations.

Summary:

- Euler Angles = Rotation specified by 3 sequential angle changes.
- Quaternions = Rotation encoded in a 4D number, better for math and computer graphics

To convert Unreal rotations to NED:

1. Convert Unreal's Euler angles (degrees) to radians
2. Adjust for Z-axis inversion (Unreal's pitch and roll signs (+/-) may need flipping depending on the drone's orientation)
3. Use AirSim's *airsim.to_quaternion(pitch, roll, yaw)* for converting Euler angles (in radians) to quaternions for API calls.

For better clarity, **AirSim's API accepts both quaternions and Euler angles**, but quaternions are the internal representation, and Euler angles are often converted to quaternions internally. For example:

- *simSetVehiclePose* requires a quaternion for orientation.
- *MoveToPositionAsync* accepts a yaw angle (Euler) via *YawMode*.

Code: In file "Part2_droneMovement_anglesRotationConversion.py"

Part 3: Control an Automated Drone

Now we want to control the drone's movement using AirSim. In the SplinePath.py file we have a move_point_to_point() function that moves the drone from a starting point to and endpoint in a 3D space.

AirSim Coordinate System:

- x: Forward/backward direction (positive x is forward).
- y: Left/right direction (positive y is right).
- z: Up/down direction (negative z is upward, positive z is downward).

Move Point to Point

This function, calls a function **interpolate_points** to generate a list of intermediate points between start and end.

By setting the **num_points=100** parameter we divide the path between the start and end points into 100 smaller segments, creating a smooth trajectory.

client.moveOnPathAsync: An AirSim API method that commands the drone to follow a sequence of points asynchronously (non-blocking). The .join() at the end makes the program wait until the movement is complete. Parameters:

- **points**: The list of 100 interpolated points for the drone to follow. (Gotten from *interpolate_points* function return)
- **velocity=5**: The drone moves at a speed of 5 meters per second along the path.
- **drivetrain=airsim.DrivetrainType.MaxDegreeOfFreedom**: Specifies the drivetrain type as MaxDegreeOfFreedom, meaning the drone can move freely in any direction (x, y, z) without constraints on its orientation or path type (e.g., it's not restricted to forward-only motion like a car).
- **yaw_mode=airsim.YawMode(is_rate=False, yaw_or_rate=0)**:
- **is_rate=False**: Indicates that *yaw_or_rate* specifies an absolute yaw angle (in degrees) rather than a yaw rate (degrees per second).
- **yaw_or_rate=0**: Sets the drone's yaw (heading) to 0 degrees, meaning the drone faces along the positive x-axis (forward) throughout the movement. This ensures the drone maintains a constant orientation (facing forward) while moving.

Starting Position:

- The drone begins at coordinates [0, 0, -2] (x=0, y=0, z=-2 meters).
- In AirSim, z=-2 means the drone is 2 meters above the ground (since negative z is upward).

Path:

- The drone moves in a **straight line** along the x-axis from x=0 to x=10, with no change in y (stays at y=0) or z (stays at z=-2).
- The path is smooth due to the 100 interpolated points, so the drone doesn't make abrupt jumps.

Orientation:

- The drone maintains a constant **yaw** of **0 degrees**, meaning it faces along the positive x-axis (forward) throughout the movement.
- The MaxDegreeOfFreedom drivetrain allows the drone to move directly along the path without needing to align its body with the direction of motion, but the yaw setting ensures it faces forward anyway.

Altitude:

- The drone maintains a constant altitude of **2 meters above the ground** (z=-2) throughout the movement.

Visual in AirSim:

- In the AirSim simulator, you'll see the drone (likely a quadcopter) take off (if not already at z=-2), then move smoothly 10 meters forward along the x-axis while staying at a fixed altitude and facing forward.
- The movement should be linear and smooth due to the interpolated points and constant velocity.

Key Points About the Movement

- **Type**: Linear point-to-point movement in 3D space.
- **Path**: Straight line from [0, 0, -2] to [10, 0, -2].
- **Distance**: 10 meters (along the x-axis).
- **Speed**: 5 meters per second (approximately 2 seconds to complete).
- **Orientation**: Fixed yaw at 0 degrees (facing forward).
- **Altitude**: Constant at 2 meters above the ground.

- **Smoothness**: The 100 interpolated points ensure a smooth trajectory rather than a single jump from start to end.

Move Multi Point Path

The *move_multi_point_path* function commands a drone in the AirSim simulator to follow a square-shaped path defined by five waypoints, returning to the starting point. Below is a concise explanation, omitting details already covered in the point-to-point movement description (e.g., interpolate_points, client.moveOnPathAsync, velocity, drivetrain, yaw_mode, and general AirSim conventions like negative z for altitude).

Path Description:

- **Waypoints**: The drone follows a square path in the xy-plane at a constant altitude, defined by:
  - [0, 0, -2]: Starting point (x=0, y=0, 2 meters above ground).
  - [10, 0, -2]: Move 10 meters along the x-axis. (go forward)
  - [10, 10, -2]: Move 10 meters along the y-axis, forming a right angle. (go right)
  - [0, 10, -2]: Move 10 meters back along the x-axis. (go backward)
  - [0, 0, -2]: Return to the starting point, completing the square. (go left)
- **Path Shape**: The waypoints form a 10x10 meter square in the xy-plane at a constant altitude of 2 meters (z=-2).
- **Movement**: The drone moves smoothly between waypoints due to 100 interpolated points per segment, creating a continuous trajectory.
- **Total Distance**: Approximately 40 meters (4 segments of 10 meters each).
- **Speed**: 5 meters per second, so the entire path takes about 8 seconds to complete.
- **Orientation**: The drone maintains a fixed yaw of 0 degrees (facing the positive x-axis) throughout, regardless of its direction of motion.
- **Visual in AirSim**: In the simulator, the drone traces a square path while staying 2 meters above the ground, moving smoothly along each side and making 90-degree turns at the corners, all while facing forward.

Move Circular Path

The *move_circular_path* function commands a drone in the AirSim simulator to follow a circular path centered at a fixed point with a specified radius and altitude. Below is a concise explanation, omitting details already covered in previous descriptions (e.g., client.moveOnPathAsync, velocity, drivetrain, yaw_mode, and general AirSim conventions like negative z for altitude).

Path Description:

Waypoints: The drone follows a circular path generated by the *generate_circle_points* function, creating 100 waypoints around a center point.

Center: [5, 5, -2] (x=5, y=5, 2 meters above ground).

Radius: 5 meters, defining the size of the circular path.

Altitude: Constant at z=-2 (2 meters above ground).

Path Shape: The waypoints form a circle in the xy-plane with a radius of 5 meters, centered at (5, 5), at a constant altitude of 2 meters.

Movement: The drone moves smoothly along the circular path due to 100 interpolated points, creating a continuous circular trajectory.

Total Distance: Approximately 31.4 meters (circumference = 2π × 5 ≈ 31.4 meters).

Speed: 2 meters per second (reduced from 5 for stability), so the entire path takes about 15.7 seconds to complete.

Move Triangular Path

The *move_triangular_path* function commands a drone in the AirSim simulator to follow a triangular path defined by four waypoints, forming an equilateral triangle and returning to the starting point.

Path Description:

- **Waypoints:** The drone follows an equilateral triangular path in the xy-plane at a constant altitude, defined by:
  - [0, 0, -2]: Starting point (x=0, y=0, 2 meters above ground).
  - [10, 0, -2]: Move 10 meters along the x-axis (first side).

  - [5, 8.66, -2]: Move to the third vertex, forming a 60-degree angle (y-coordinate ≈ 10 √3/2 for an equilateral triangle). → $5^2 + 8.66^2 = 10^2$ → 25 + 74.9956 = 99.9956
  - [0, 0, -2]: Return to the starting point, completing the triangle.
- **Path Shape**: The waypoints form an equilateral triangle in the xy-plane with side length 10 meters at a constant altitude of 2 meters (z=-2).
- **Movement**: The drone moves smoothly between waypoints due to 100 interpolated points per segment, creating a continuous triangular trajectory.
- **Total Distance**: Approximately 30 meters (3 sides of 10 meters each).
- **Speed**: 5 meters per second, so the entire path takes about 6 seconds to complete.
- **Orientation:** With drivetrain=airsim.DrivetrainType.MaxDegreeOfFreedom and yaw_mode=airsim.YawMode(is_rate=False, yaw_or_rate=0), the drone maintains a fixed yaw of 0 degrees (facing the positive x-axis) throughout, regardless of its direction of motion.
- **Visual in AirSim**: In the simulator, the drone traces an equilateral triangular path at 2 meters above the ground, moving smoothly along each side with 60-degree turns at the vertices, while consistently facing forward (positive x-axis).

Move Half-Circle Path

The *move_half_circle_path* function commands a drone in the AirSim simulator to follow a half-circle path centered at a fixed point with a specified radius and altitude.

Path Description:

- **Waypoints:** The drone follows a half-circle path generated by the generate_half_circle_points function, creating 50 waypoints forming a 180-degree arc around a center point.
  - Center: [5, 5, -2] (x=5, y=5, 2 meters above ground).
  - Radius: 5 meters, defining the size of the half-circle.
  - Altitude: Constant at z=-2 (2 meters above ground).
- **Path Shape:** The waypoints form a semicircular arc in the xy-plane, starting and ending on opposite sides of the circle centered at (5, 5), at a constant altitude of 2 meters.
- **Movement**: The drone moves smoothly along the half-circle due to 50 interpolated points, creating a continuous semicircular trajectory.
- **Total Distance**: Approximately 15.7 meters (half the circumference of a circle with radius 5, i.e., π × 5 ≈ 15.7 meters).

- **Speed**: 5 meters per second, so the entire path takes about 3.14 seconds to complete.
- **Visual in AirSim**: In the simulator, the drone traces a smooth 180-degree arc centered at (5, 5), staying 2 meters above the ground, moving from one side of the circle to the opposite side while consistently facing forward (positive x-axis).

Code: In file “DroneControl_SplinePaths.py”