

AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles

Shital Shah¹, Debadepta Dey², Chris Lovett³, Ashish Kapoor⁴

Abstract Developing and testing algorithms for autonomous vehicles in real world is an expensive and time consuming process. Also, in order to utilize recent advances in machine intelligence and deep learning we need to collect a large amount of annotated training data in a variety of conditions and environments. We present a new simulator built on Unreal Engine that offers physically and visually realistic simulations for both of these goals. Our simulator includes a physics engine that can operate at a high frequency for real-time hardware-in-the-loop (HITL) simulations with support for popular protocols (e.g. MavLink). The simulator is designed from the ground up to be extensible to accommodate new types of vehicles, hardware platforms and software protocols. In addition, the modular design enables various components to be easily usable independently in other projects. We demonstrate the simulator by first implementing a quadrotor as an autonomous vehicle and then experimentally comparing the software components with real-world flights.

1 Introduction

Recently, paradigms such as reinforcement learning [12], learning-by-demonstration [2] and transfer learning [25] are proving a natural means to train various robotics systems. One of the key challenges with these techniques is the high sample complexity - the amount of training data needed to learn useful behaviors is often prohibitively high. This issue is further exacerbated by the fact that autonomous vehicles are often unsafe and expensive to operate during the training phase. In order to seamlessly operate in the real world the robot needs to transfer the learning it does in simulation. Currently, this is a non-trivial task as simulated perception, environments and actuators are often simplistic and lack the richness or diversity of the real world. For example, for robots that aim to use computer vision in outdoor en-

1, 2, 3, 4, Microsoft Research, Redmond, WA, USA e-mail: shital.s, dedey, clovett, akapoor@microsoft.com

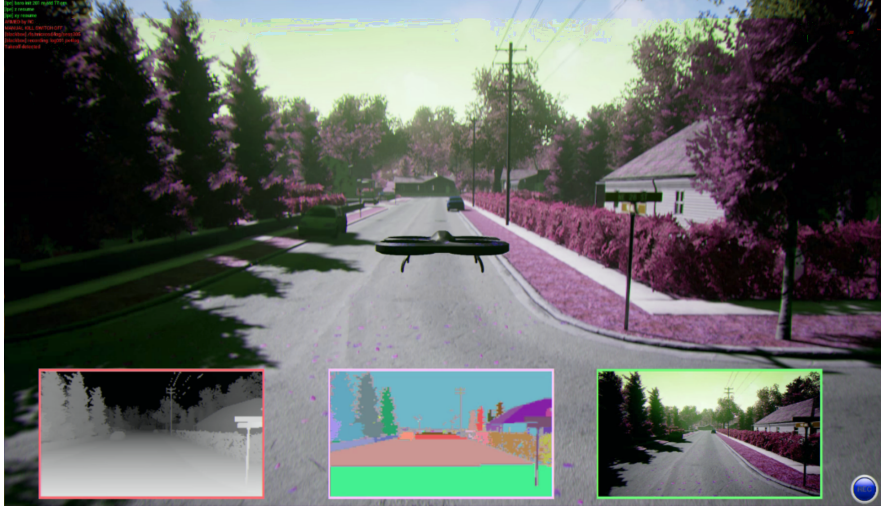


Fig. 1 A snapshot from AirSim shows an aerial vehicle flying in an urban environment. The inset shows depth, object segmentation and front camera streams generated in real time.

vironments, it may be important to model real-world complex objects such as trees, roads, lakes, electric poles and houses along with rendering that includes finer details such as soft shadows, specular reflections, diffused inter-reflections and so on. Similarly, it is important to develop more accurate models of system dynamics so that simulated behavior closely mimics the real-world.

AirSim is an open-source platform [21] that aims to narrow the gap between simulation and reality in order to aid development of autonomous vehicles. The platform seeks to positively influence development and testing of data-driven machine intelligence techniques such as reinforcement learning and deep learning. It is inspired by several previous simulators (see related work), and one of our key goals is to build a community to push the state-of-the-art towards this goal.

2 Related Work

While an exhaustive review of currently used simulators is beyond the scope of this paper, we mention a few notable recent works that are closest to our setting and has deeply influenced this work.

Gazebo [13] has been one of the most popular simulation platforms for the research work. It has a modular design that allows to use different physics engines, sensor models and create 3D worlds. Gazebo goes beyond monolithic rigid body vehicles and can be used to simulate more general robots with links-and-joints architecture such as complex manipulator arms or biped robots. While Gazebo is fairly feature rich it has been difficult to create large scale complex visually rich environments

that are closer to the real world and it has lagged behind various advancements in rendering techniques made by platforms such as Unreal engine or Unity.

Other notable efforts includes Hector [17] that primarily focuses on tight integration with popular middleware ROS and Gazebo. It offers wind tunnel tuned flight dynamics, sensor models that includes bias drift using Gaussian Markov process and software-in-loop using Orocos toolchain. However, Hector lacks support for popular hardware platforms such as Pixhawk and protocols such as MavLink. Because of its tight dependency on ROS and Gazebo, it's limited by richness of simulated environments as noted previously.

Similarly, RotorS [7] provides a modular framework to design Micro Aerial Vehicles, and build algorithms for control and state estimation that can be tested in simulator. It is possible to setup RotorS for HITL with Pixhawk. RotorS also uses Gazebo as its platform, consequently limiting its perception related capabilities.

Finally, jMavSim [1] is easy to use simulator that was designed with a goal of testing PX4 firmware and devices. It is therefore tightly coupled with PX4 simulation APIs, uses albeit simpler sensor models and utilizes simple rendering engine without any objects in the environment.

Apart from these, there have been many games like simulators and training applications, however, these are mostly commercial closed-source software with little or no public information on models, accuracy of simulation or development APIs for autonomous applications.

3 Architecture

Our simulator follows a modular design with an emphasis on extensibility. The core components includes environment model, vehicle model, physics engine, sensor models, rendering interface, public API layer and an interface layer for vehicle firmware as depicted in Figure 2.

The typical setup for an autonomous aerial vehicle includes the flight controller firmware such as PX4 [16], ROSFlight [10], Hackflight[15] etc. The flight controller takes desired state and the sensor data as inputs, computes the estimate of current state and outputs the actuator control signals to achieve the desired state. For example, in case of quadrotors, user may specify desired pitch, roll and yaw angles as desired state and the flight controller may use sensor data from accelerometer and gyroscope to estimate the current angles and finally compute the motor signals to achieve the desired angles.

During simulation, the simulator provides the sensor data from the simulated world to the flight controller. The flight controller outputs the actuator signals which is taken as input by the the vehicle model component of the simulator. The goal of the vehicle model is to compute the forces and torques generated by the simulated actuators. For example, in case of quadrotors, we compute the thrust and torques produced by the propellers given the motor voltages. In addition, there may be forces generated from drag, friction and gravity. These forces and torques are then taken

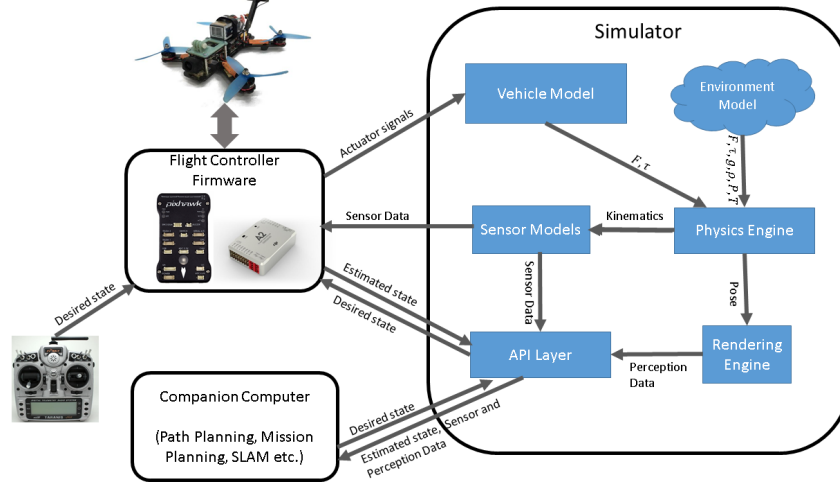


Fig. 2 The architecture of the system that depicts the core components and their interactions.

as inputs by the physics engine to compute the next kinematic state of bodies in the simulated world. This kinematic state of bodies along with the environment models for gravity, air density, air pressure, magnetic field and geographic location (GPS coordinates) provides the ground truth for the simulated sensor models.

The desired state input to the flight controller can be set by human operator using remote control or by a companion computer in the autonomous setting. The companion computer may perform expensive higher level computations such as determining next desired waypoint, performing simultaneous localization and mapping (SLAM), computing desired trajectory etc. The companion computer may have to process large amount of data generated by the sensors such as vision cameras and lidars which in turn requires that simulated environments have reasonable details. This has been one of the challenging areas where we leverage recent advances in rendering technologies implemented by platforms such as Unreal engine [11]. In addition, we also utilize the underlying pipeline in the Unreal engine to detect collisions. The companion computer interacts with the simulator via a set of APIs that allows it to observe the sensor streams, vehicle state and send commands. These APIs are designed such that it shields the companion computer from being aware of whether its being run under simulation or in the real world. This is particularly important so that one can develop and test algorithms in simulator and deploy to real vehicle without having to make additional changes.

The AirSim code base is implemented as a plugin for the Unreal engine that can be dropped in to any Unreal project. The Unreal engine platform offers an elaborate marketplace with hundreds of pre-made detailed environments, many created using photogrammetry techniques [18] to generate reasonably faithful reconstruction of real-world scenes.

Next, we provide more details on the individual components of the simulator.

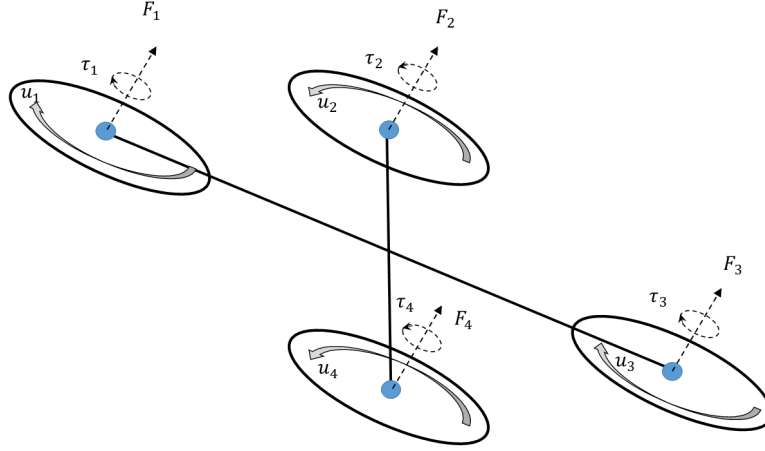


Fig. 3 Vehicle model for the quadrotor. The four blue vertices experience the controls u_1, \dots, u_4 , which in turn results in the forces $\mathbf{F}_1, \dots, \mathbf{F}_4$ and the torques τ_1, \dots, τ_4 .

3.1 Vehicle Model

AirSim provides an interface to define vehicle as a rigid body that may have arbitrary number of actuators generating forces and torques. The vehicle model includes parameters such as mass, inertia, coefficients for linear and angular drag, coefficients of friction and restitution which is used by the physics engine to compute rigid body dynamics.

Formally, a vehicle is defined as a collection of K vertices placed at positions $\{\mathbf{r}_1, \dots, \mathbf{r}_k\}$ and normals $\{\mathbf{n}_1, \dots, \mathbf{n}_k\}$, each of which experience a unitless vehicle specific scalar control input $\{u_1, \dots, u_k\}$. The forces and torques from these vertices are assumed to be generated in the direction of their normals. However note that the positions as well as normals are allowed to change during the simulation.

Figure 3 shows how a quadrotor can be depicted as a collection of four vertices. The control input u_i drives the rotational speed of the propellers located at the four vertices. We compute the forces and torques produced by propellers using [4]:

$$\mathbf{F}_i = C_T \rho \omega_{max}^2 D^4 u_i \quad \text{and} \quad \tau_i = \frac{1}{2\pi} C_{pow} \rho \omega_{max}^2 D^5 u_i.$$

Here C_T and C_{pow} are the thrust and the power coefficients respectively and are based on the physical characteristics of the propeller, ρ is the air density, D is the propeller's diameter and ω_{max} is the max angular velocity in revolutions per minute. By allowing the movements of these vertices during the flight it is possible to simulate the vehicles with capabilities such as Vertical Take-Off and Landing (VTOL) and other recent quadrotors that change their configuration in flight.

The vehicle model abstract interface also provides a way to specify the cross sectional area in body frame that in turn can be used by physics engine to compute the linear and angular drag on the body.

3.2 Environment

The vehicle is exposed to various physical phenomena including gravity, air-density, air pressure and magnetic field. While it is possible to produce computationally expensive models of these phenomena that are very accurate, we focus our attention to models that are accurate enough to allow a real-time operation with hardware-in-the-loop. We describe these individual components of the environment below.

3.2.1 Gravity

While many models use a constant number to model the gravity, it varies in a complex manner as demonstrated by models such as GRACE [23]. For most ground based or low altitude vehicles these variations may not be important; however, it is fairly inexpensive to incorporate a more accurate model. Formally, we approximate the gravitational acceleration g at height h by applying binomial theorem on Newton's law of gravity and neglecting the higher powers:

$$g = g_0 \cdot \frac{R_e^2}{(R_e + h)^2} \approx g_0 \cdot \left(1 - 2\frac{h}{R_e}\right).$$

Here R_e is Earth's radius and g_0 is the gravitational constant measured at the surface.

3.2.2 Magnetic Field

Accurately modeling the magnetic field of a complex body such as Earth is a computationally expensive task. The World Magnetic Model (WMM) model [6] by National Oceanic and Atmospheric Administration (NOAA) is one of the best known magnetic models of Earth. Unfortunately, the most recent model WMM2015 is fairly complex and computationally expensive for real-time applications.

We implemented the tilted dipole model where we assume Earth as a perfect dipole sphere [14, pp 27-30]. This ignores all but the first order terms to derive magnetic field estimate using the spherical geometry. This model allows us to simulate variation of the magnetic field as we move in space as well as areas that are often problematic such as polar regions. Given a geographic latitude θ , longitude ϕ and altitude h (from surface of the earth), we first compute the magnetic co-latitude θ_m using:

$$\cos \theta_m = \cos \theta \cos \theta^0 + \sin \theta \sin \theta^0 \cos(\phi - \phi^0).$$

Where θ^0 and ϕ^0 denote the latitude and longitude of the true magnetic north pole. Then, the total magnetic intensity $|B|$ is computed as:

$$|B| = B_0 \left(\frac{R_e}{R_e + h}\right)^3 \sqrt{1 + 3 \cos^2 \theta_m}$$

Here B_0 is the mean value of the magnetic field at the magnetic equator on the Earth's surface, θ_m is the magnetic co-latitude and R_e is the mean radius of the Earth. Next, we determine the inclination α and declination β angles using:

$$\tan \alpha = 2 \cot \theta_m \quad \text{and} \quad \sin \beta = \begin{cases} \sin(\phi - \phi^0) \frac{\cos \theta^0}{\sin \theta_m}, & \text{if } \cos \theta_m > \sin \theta^0 \sin \theta \\ \cos(\phi - \phi^0) \frac{\cos \theta^0}{\sin \theta_m}, & \text{otherwise.} \end{cases}$$

Finally, we can compute the horizontal field intensity (H), the latitudinal (X), the longitudinal (Y) and the vertical field (Z) components of the magnetic field vector as follows:

$$H = |B| \cos \alpha \quad Z = |B| \sin \alpha \quad X = H \cos \beta \quad Y = H \sin \beta.$$

3.2.3 Air Pressure and Density

The relationship between the altitude and the pressure of the Earth's atmosphere is complicated due to the presence of many distinct layers, each with its own individual properties. First we compute Standard Temperature T and Standard Pressure P using 1976 U.S. Standard Atmosphere model [22, eq 1.16, 1.17] for altitude below 51 kilometers and switch to the model in [3, Table 4] beyond that up to 86 km. Then, the air density is $\rho = \frac{P}{R \cdot T}$ (where R is the specific gas constant.)

3.3 Physics Engine

The kinematic state of the body is expressed using 6 quantities: position, orientation, linear velocity, linear acceleration, angular velocity and angular acceleration. The goal of the physics engine is to compute the next kinematic state for each body given the forces and torques acting on it. We strive for an efficient physics engine that can run its update loop at high frequency (1000 Hz) which is desirable for enabling real-time simulation scenarios such as high speed quadrotor control. Consequently, we implement a physics engine that avoids the extra complexities of a generic engine allowing us to tightly control the performance and make trade-offs that best meet our requirements.

3.3.1 Linear and Angular drag

Since the vehicle moves in the presence of air, the linear and the angular drag has a significant effect on the dynamics of the body. The simulator computes the magnitude $|\mathbf{F}_d|$ of the linear drag force on the body according to the drag equation [24]:

$$|\mathbf{F}_d| = \frac{1}{2} \rho |\mathbf{v}|^2 C_{lin} A.$$

Here C_{lin} is the linear air drag coefficient, A is the vehicle cross-section and ρ is the air density. This drag force acts in the direction opposite to the velocity vector \mathbf{v}

Computing the angular drag for arbitrary shape remains complex and computationally intensive task. Many existing physics engines use a small but often an arbitrary damping constant as a substitute for computing actual angular drag. We provide simple but better approximations to model the angular drag.

Consider an infinitesimal surface area ds in the extremity of the body experiencing the angular velocity $\boldsymbol{\omega}$. As the linear velocity $d\mathbf{v}$ experienced by ds is given by $\mathbf{r}_{ds} \times \boldsymbol{\omega}$, we can now use the linear drag equation for ds [19, pp 160-161]:

$$|d\mathbf{F}| = \frac{1}{2}\rho|\mathbf{r}_{ds} \times \boldsymbol{\omega}|^2 C_{lin} ds, \quad \text{where direction of } d\mathbf{F} \text{ is } -\mathbf{r}_{ds} \times \boldsymbol{\omega}.$$

Now, the drag torque is computed by integrating over the entire surface: $\boldsymbol{\tau}_d = \int_S \mathbf{r}_{ds} \times d\mathbf{F}$. To simplify the implementation, we approximate the body of the vehicle as set of connected faces which further can be approximated as a rectangular box for the purpose of evaluating the integral.

3.3.2 Accelerations

In addition to the drag forces and torques, we also need to consider the forces \mathbf{F}_i and the torques $\boldsymbol{\tau}_i$ present on the vehicle at the vertex located at \mathbf{r}_i relative to center of gravity (see section 3.1). We thus compute the net force and torque as:

$$\mathbf{F}_{net} = \sum_i \mathbf{F}_i + \mathbf{F}_d \quad \text{and} \quad \boldsymbol{\tau}_{net} = \sum_i [\boldsymbol{\tau}_i + \mathbf{r}_i \times \mathbf{F}_i] + \boldsymbol{\tau}_d.$$

We obtain the linear acceleration by applying Newton's second law and then adding gravity vector to compute the net acceleration, $\mathbf{a} = \mathbf{F}_{net}/m + \mathbf{g}$. The angular acceleration in body frame is given by Euler's rotation equation: $\boldsymbol{\alpha} = I^{-1} \cdot (\boldsymbol{\tau}_{net} - (\boldsymbol{\omega} \times (I \cdot \boldsymbol{\omega})))$, where, I is the inertia tensor and $\boldsymbol{\omega}$ is angular velocity, both in body frame.

3.3.3 Integration

We update the position \mathbf{p}_{k+1} of the body at time $k+1$ by integrating the velocity and the initial position \mathbf{p}_0 . The first order integration algorithms such as Euler method diverges quickly with unbounded error although very simple to implement. In our implementation we use Velocity Verlet algorithm instead of Runge Kutta for its computationally inexpensiveness and stability while still being second order method [9]. Formally,

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \frac{\mathbf{a}_k + \mathbf{a}_{k+1}}{2} \cdot dt \quad \mathbf{p}_{k+1} = \mathbf{p}_k + \mathbf{v}_k \cdot dt + \frac{1}{2} \cdot \mathbf{a}_k \cdot dt^2$$

The angular velocity is updated in similar manner as linear velocity however updating orientation isn't straight forward. One of the approach is to maintains the orientation as a rotation matrix that is updated every time step. However this causes a slow drift which must be corrected by orthonormalization at regular intervals which is expensive. Alternative approach is to maintain rotations as much more efficient quaternions which are also numerically stable and trivially normalizable. One of the problem, however, is that the orientation quaternion is maintained in the world frame while the angular velocity is maintained in the body frame in our framework. To update the orientation, we first compute the angle-axis pair $(\alpha_{dt}, \mathbf{u})$ where α_{dt} is the angle traversed around unit vector \mathbf{u} . We can compute the angle $\alpha_{dt} = |\boldsymbol{\omega}| \cdot dt$ and axis by $u = \boldsymbol{\omega}/|\boldsymbol{\omega}|$. This allows us to compute equivalent change in quaternion \mathbf{q}_{dt} representing the change in orientation in time dt . As noted before, \mathbf{q}_{dt} is in body frame while \mathbf{q}_k in world reference frame. The problem now remains that of adding \mathbf{q}_{dt} to \mathbf{q}_k to obtain \mathbf{q}_{k+1} which can be proven to given by relationship $\mathbf{q}_{k+1} = \mathbf{q}_k \cdot \mathbf{q}_{dt}$.

3.3.4 Collisions

Unreal engine offers a rich collision detection system optimized for different classes of collision meshes and we directly use this feature for our needs. We receive the impact position, impact normal and penetration depth for each collision that occurred during the render interval. Our physics engine uses this data to compute the collision response with Coulomb friction to modify both linear and angular kinematics.[8]

3.4 Sensors

AirSim offers sensor models for accelerometer, gyroscope, barometer, magnetometer and GPS. All our sensor models are implemented as C++ header-only library and can be independently used outside of AirSim. Like other components, sensor models are expressed as abstract interfaces so it is easy to replace or add new sensors.

3.4.1 Barometer

To simulate barometer, we compute ground truth pressure using the detailed model of atmosphere (sec 3.2.3) and model the drift in the pressure measurement over time using Gaussian Markov process [20] for more realistic behavior in long flights. Formally, if we denote the current bias factor as b_k then the drift is modeled as:

$$b_{k+1} = w \cdot b_k + (1 - w) \cdot \eta, \text{ where: } w = e^{-\frac{dt}{\tau}} \text{ and } \eta \sim N(0, s^2).$$

Here τ , is the time constant for the process and set to 1 hour in our model. η is a zero mean Gaussian noise with standard deviation that can be selected using the

data available in [5]. This pressure p is then added with white noise drawn from zero mean Gaussian distribution with standard deviation set from datasheet of the sensor (such as MEAS MS56112). Finally we convert the pressure to altitude using barometric formula used by the sensor's driver:

$$h = \frac{T_0}{a} \left[\left(\frac{p}{p_0} \right)^{-\left(\frac{aR}{g}\right)} - 1 \right],$$

here T_0 is the reference temperature (15 deg C), $a = -6.5 \times 10^{-3}$ is the temperature gradient, g and R are the gravity and the specific gas constants, p_0 is the current sea level pressure and p is the measurement.

3.4.2 Gyroscope and Accelerometer

Gyroscope and accelerometers constitute the core of the inertial measurement unit (IMU) [26]. We model these by adding white noise and bias drift over time to the ground truth. For gyroscope, given the true angular velocity in body frame ω , we compute the measurement ω^{out} as,

$$\begin{aligned} \omega^{\text{out}} &= \omega + \eta_a + b_t, & \text{where } \eta_a &\sim N(0, r_a) \text{ and} \\ b_t &= b_{t-1} + \eta_b, & \text{where } \eta_b &\sim N\left(0, b_0 \sqrt{\frac{dt}{t_a}}\right). \end{aligned}$$

Here parameters r_a , bias b_0 and the time constant for bias drift t_a can either be obtained from Allan variance plots or from datasheets. Accelerometer output is computed in the similar manner except that we must first subtract gravity from the true linear acceleration in the world frame and then convert the result to the body frame before we add bias drift and noise.

3.4.3 Magnetometer

We use the tilted dipole model for Earth's magnetic field 3.2.2, given the geographic coordinates to compute the components of the ground truth magnetic field in body frame and add the white noise as specified in the datasheet.

3.4.4 Global Positioning System (GPS)

Our GPS model simulates latency (typically 200ms), slower update rates (typically 50 Hz) and horizontal and vertical position error estimate decay rates to simulate gaining fix over time. The decay rate is modeled using first order low pass filter individually parameterized for horizontal and vertical fix.

3.5 Visual Rendering

Since advanced rendering and detailed environments have been a key requirement for AirSim we chose Unreal Engine 4 (UE4) [11] as our rendering platform. UE4 offers several features that made it an attractive choice including it being an open source and available on Linux, Windows as well as OSX. UE4 brings some of the cutting edge graphics features such as physically based materials, photometric lights, planar reflections, ray traced distance field shadows, lit translucency etc. Figure 1 shows a screen-shot from AirSim which highlight near photo-realistic rendering capabilities. Further, Unreal’s large online Marketplace has various pre-made elaborate environments, many of which are created using photogrammetry techniques.

4 Experiments

We perform experiments primarily to evaluate how close the flight characteristic of a quadrotor flying in real-world is to that of a simulation of the same vehicle in AirSim. We also evaluate some of our sensor models against the real-world sensors.

Hardware Platform: Real-world flights were performed with the Pixhawk v2 flight controller mounted on a Flamewheel quadrotor frame, together with a Gigabyte 5500 Brix running Ubuntu 16.04. The sensor measurements were recorded on the Pixhawk device itself. We configured the simulated quadrotor in AirSim using the measured physical parameters and simulated sensor models configured using sensor data sheets. The AirSim MavLinkTest application was used to perform repeatable offboard control for both the real-world and the simulated flights.

Trajectory Evaluation: We fly the quadrotor in the simulator in two different patterns: (1) trajectory in square shape with each side being 5m long (2) trajectory in circle shape with radius being 10m long. We then use exact same commands to fly the real vehicle. For both the simulation and the real-world flights, we collect location of the vehicle in local NED coordinates along with timestamps.

Figure 4(c) and 4(d) shows the time series of locations in simulated flight and the real flight. Here, the horizontal axis represents the time and the vertical axis represent the off-set in X and Y directions. We also compute the symmetric Hausdorff distance between the real-world track and the track in simulation. We found that the simulation and real-world tracks were fairly close both for the circle (Hausdorff distance between simulated and real-world: 1.47 m) as well as the square (Hausdorff distance between simulated and real-world: 0.65 m).

We also present visual comparison for this experiment for the circle and the square patterns in Figures 4(a) and 4(b) respectively. The simulated trajectory is shown with a purple line while the real trajectory is shown with a red line. We can observe that qualitatively the trajectories tracked by both the real-world and the simulated vehicle are close. The small differences may have been caused by various

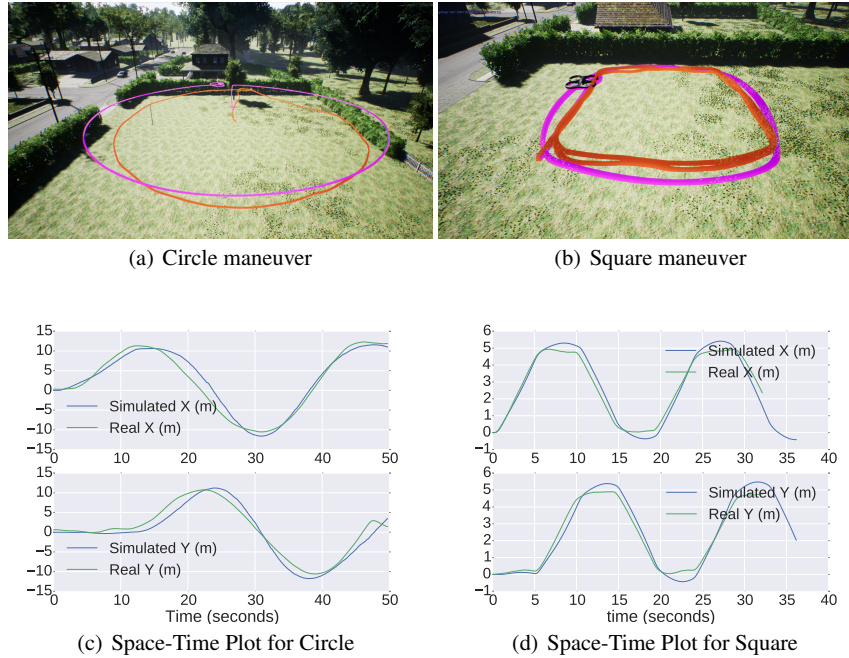


Fig. 4 Evaluating the differences between the simulated and the real-world flight. In top figures, the purple and the red lines depict the track from simulation and the real-world flights respectively.

factors such as integration errors, vehicle model approximations and mild random winds.

Sensor Models: Besides evaluating the entire simulation pipeline we also investigated individual component models, namely the barometer (MEAS MS5611-01BA), the magnetometer (Honeywell HMC5883) and the IMU (InvenSense MPU 6000). Note that the simulated GPS model is currently simplistic, thus, we only focus on the three more complex sensor models. For each of the above sensors we use the manufacture specified datasheets to set the parameters in the sensor models.

- **IMU:** We measured readings from the accelerometers and gyroscope as the vehicle was stationary and flying. We observed that while the characteristics were similar when the vehicle was stationary (gyro: simulated variance $2.47e-7$ rad^2/s^2 , real-world variance $6.71e-7$ rad^2/s^2 , accel.: simulated variance $1.78e-4$ m^2/s^4 , real-world variance $1.93e-4$ m^2/s^4), the observed variance for an in-flight vehicle was much higher than the simulated one (accel.: simulated $1.75e-3$ m^2/s^4 vs. real-world 9.46 m^2/s^4). This is likely in real-world the airframe vibrates when the motors are running and that phenomenon is not yet modeled in AirSim.
- **Barometer:** We raised the sensor periodically between two fixed heights: ground level and then elevated to 178 cm (both in simulation and real-world). Figure 5(a) shows both the measurements (green is simulated, blue is real-world) and we ob-

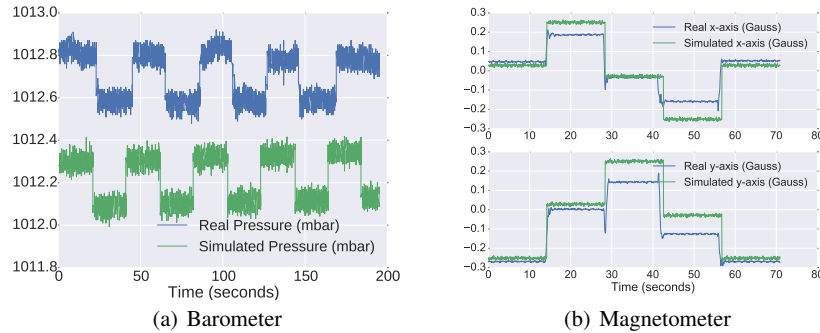


Fig. 5 Figure 5(a) and 5(b) show that barometer and the magnetometer characteristics in simulation closely resemble that of the real world.

serve that the signals have similar characteristics. Note that the offset between the simulated and the real-world pressure is due the difference in absolute pressure in the real-world and the one in the simulation. There is also a small increase in the middle due to a temperature increase, which wasn't simulated. Overall, the characteristics of the simulated sensor matches well to the real sensor.

- **Magnetometer:** We placed the vehicle on the ground and then rotated it by 90° four times. Figure 5(b) shows the real-world and the simulated measurements and highlight that they are very similar in characteristic.

5 Conclusion and Future Work

AirSim offers hi-fidelity physical and visual simulation that allows to generate large quantity of training data cheaply for building machine learning models. AirSim API design allows developing algorithms against simulator and then deploy them without change on real vehicles. The core components of AirSim including physics engine, vehicle models, environment models and sensor models are designed to be independently usable with minimal dependencies outside of AirSim and are easily extensible. AirSim is inspired by the goal of developing reinforcement learning algorithms for the autonomous agents that can operate in the real world.

The task of mimicking the real-world in *real-time simulation* is a challenging endeavor. There are a number of things that can be improved. Currently we do not simulate richer collision response or advanced ground interaction models which may be possible in future by implementing our physics engine interface with NVIDIA PhysX and utilizing features such as physics sub-stepping. Also we do not simulate various oddities in camera sensors except those directly available in Unreal engine. We plan to add advanced noise models and lens models in future. The degradation of GPS signal due to obstacles is not simulated yet which we plan to add using ray tracing methods. We also plan to add more advanced wind effects and thermal

simulations for fixed wing vehicles. Our extensibility APIs have been designed with above future work in mind and can also be used to realize other vehicle types.

References

1. Babushkin, A.: Jmavsim. <https://pixhawk.org/dev/hil/jmavsim>
2. Bagnell, J.A.: An invitation to imitation. Tech. rep., CMU ROBOTICS INST (2015)
3. Braeunig, R.: Atmospheric models. <http://www.braeunig.us/space/atmmodel.htm> (2014)
4. Brandt, J., Deters, R., Ananda, G., Selig, M.: Uiuc propeller database, university of illinois at urbana-champaign. <http://m-selig.ae.illinois.edu/props/propDB.html> (2015)
5. Burch D., B.T.: Mariner's Pressure Atlas: Worldwide Mean Sea Level Pressures and Standard Deviations for Weather Analysis. Starpath School of Navigation (2014)
6. Chulliat, A., Macmillan, S., Alken, P., Beggan, C., Nair, M., Hamilton, B., Woods, A., Ridley, V., Maus, S., Thomson, A.: The us/uk world magnetic model for 2015-2020 (2015). DOI 10.7289/V5TB14V7
7. Furrer, F., Burri, M., Achtelik, M., Siegwart, R.: Rotorsa modular gazebo mav simulator framework. In: Robot Operating System (ROS), pp. 595–625. Springer (2016)
8. Hecker, C.: Physics, part 3: Collision response. Game Developer Magazine (1997)
9. Herman, R.: A first course in differential equations for scientists and engineers. <http://people.uncw.edu/hermanr/mat361/ODEBook/> (2017)
10. Jackson, J., Ellingson, G., McLain, T.: Rosflight: A lightweight, inexpensive mav research and development tool. In: ICUAS, pp. 758–762 (2016). DOI 10.1109/ICUAS.2016.7502584
11. Karis, B., Games, E.: Real shading in unreal engine 4. In: Proc. Physically Based Shading Theory Practice (2013)
12. Kober, J., Bagnell, J.A., Peters, J.: Reinforcement learning in robotics: A survey. Int. J. Rob. Res. **32**(11), 1238–1274 (2013). DOI 10.1177/0278364913495721
13. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IROS (2004)
14. Lanza, R., Meloni, A.: The Earth's Magnetism: An Introduction for Geologists. Springer Science & Business Media (2006)
15. Levy, S.: Hackflight: Simple quadcopter flight control firmware and simulator for c++ hackers. <https://github.com/simondlevy/hackflight>
16. Meier, L., Tanskanen, P., Fraundorfer, F., Pollefeys, M.: Pixhawk: A system for autonomous flight using onboard computer vision. In: ICRA, pp. 2992–2997. IEEE (2011)
17. Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., Von Stryk, O.: Comprehensive simulation of quadrotor uavs using ros and gazebo. In: SIMPAR, pp. 400–411. Springer (2012)
18. Moore, H.: Creating assets for the open world demo (2015)
19. Nakayama, Y., Boucher, R.: Introduction to fluid mechanics. Butterworth-Heinemann (1998)
20. Sabatini, A.M., Genovese, V.: A stochastic approach to noise modeling for barometric altimeters. Sensors (Basel, Switzerland) **13**(11), 15,692–15,707 (2013)
21. Shah, S., Dey, D., Lovett, C., Kapoor, A.: Airsim open source platform at github. <https://github.com/Microsoft/AirSim> (2017)
22. Stull, R.: Practical Meteorology: An Algebra-based Survey of Atmospheric Science. University of British Columbia (2015)
23. Tapley, B., Ries, J., Bettadpur, S., Chambers, D., Cheng, M., Condi, F., Poole, S.: The ggm03 mean earth gravity model from grace. In: American Geophysical Union, G42A-03 (2007)
24. Taylor, J.: Classical mechanics. University Science Books (2005)
25. Weiss, K., Khoshgoftar, T.M., Wang, D.: A survey of transfer learning. Journal of Big Data **3**(1), 9 (2016). DOI 10.1186/s40537-016-0043-6
26. Woodman, O.J.: An introduction to inertial navigation. Tech. Rep. UCAM-CL-TR-696, University of Cambridge, Computer Laboratory (2007)