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

Shital Shah[1], Debadeepta Dey[2], Chris Lovett[3], Ashish Kapoor[4]

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 [18], learning-by-demonstration [14, 9] and transfer learning [23] 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 out-

1, 2, 3, 4, Microsoft Research, Redmond, WA, USA e-mail: shitals,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.

door environments, 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 etc. 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 (GitHub: [2]) 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 (e.g. reinforcement learning, deep learning etc.). 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 deeply influenced this work.

Gazebo [19] has been one the most popular simulation platforms. It has a wide range of capabilities and modular design including the ability to use different physics engines, variety of sensor models and ability to create 3D virtual 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 environments that are closer to the real world or use platforms such as Unreal engine or Unity as its rendering layer natively.

Other notable effort includes Hector [21] that primarily focuses on ROS and is a Gazebo-enabled quadrotor simulator. It offers wind tunnel tuned flight dynamics, sensor models that includes bias drift using Gaussian Markov process and closed-loop control using Orocos toolchain. However, Hector lacks support for popular hardware platforms such as Pixhawk and protocols such as MavLink. It is also limited by richness of available environments in Gazebo as noted previously.

Similarly, RotorS [13] provides a modular framework to design Micro Aerial Vehicles, and build algorithms for control and state estimation that can be deployed in the field. 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 [4] is one of the easiest simulators to setup and use for HITL/SITL with PX4 firmware devices. It uses albeit simpler models and is more tightly coupled with PX4/MavLink while using its own simple rendering engine.

Apart from these, there have been many games like simulator 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 include the environment model, the vehicle model, the physics engine, the sensor models, the rendering interface, API layer and the low level controller layer as exposed by vehicle firmware as depicted in Figure 2. These components are implemented as a C++ header-only library and the core functionality is exposed via APIs accessible using remote procedure calls.

The typical setup for an autonomous aerial vehicle includes the flight controller software such as PX4 [20], ROSFlight [15] etc. This flight controller runs a real-time control loop that receives the sensor data and then generates the control signals for the actuators in order to achieve the target desired state. For instance, in case of quadrotors, the typical sensor data provided to flight controller may be gyroscope and accelerometer, the desired state can be specified in terms of either pitch, roll and yaw levels or rate and output of the flight controller may be Pulse-Width-Modulation (PWM) signals that drives the motors.

During simulation, the simulator provides sensor data to the flight controller instead of that data coming from the real world. The vehicle model component of the simulator consumes the control signals generated by the flight controller and it uses it to compute the forces and torques generated by the simulated actuators. The physics engine then takes these forces and torques along with any others acting on the vehicle from external environment to compute its next kinematic state. At each time step, we are interested in knowing the position, the orientation and lin-

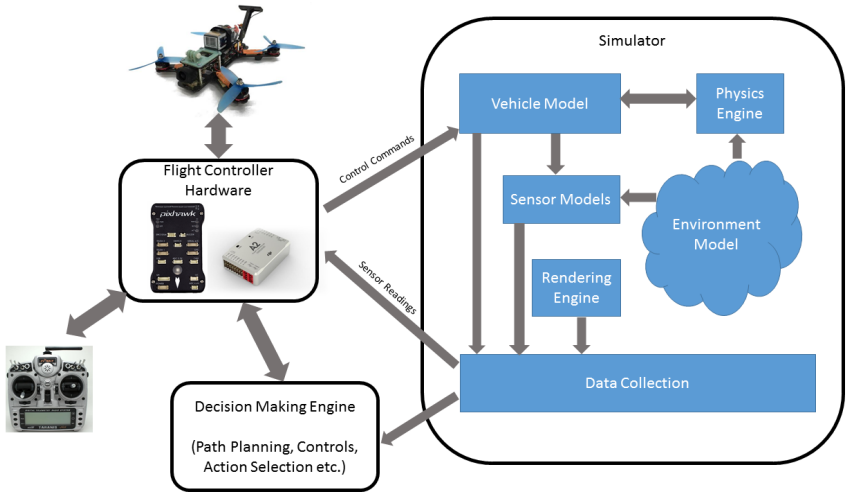


Fig. 2 The architecture of the system that depicts the core components and their interactions.

ear as well as angular velocities and accelerations. These quantities along with the environment specific models for gravity, air density, pressure, magnetic field and geographic location sets up the ground truth that drive the simulated sensor models.

The desired state for 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 more expensive computation such as simultaneous localization and mapping (SLAM), mission planning etc. to compute the desired next state for the flight controller. 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 [17]. In addition, we also utilize the underlying pipeline in the Unreal engine to detect collisions. The companion computer interacts with the simulation via a set of APIs that allows it to observe the sensor streams and send commands to the flight controller. 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 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 with many created using photogrammetry techniques [6] to generate reasonably faithful reconstruction of real-world scenes.

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

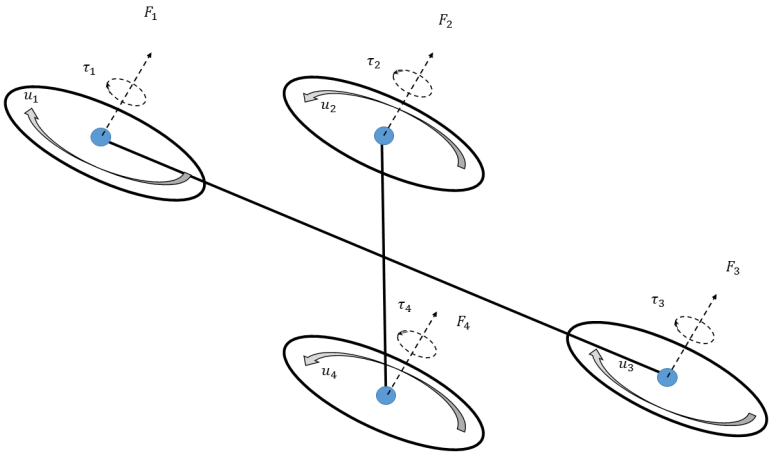


Fig. 3 Vehicle model for the quadrotor. The four blue vertices experience the controls $u_1,..u_4$, which in turn results in the forces $\mathbf{F}_1,..,\mathbf{F}_4$ and the torques $\tau_1,..,\tau_4$.

3.1 Vehicle Model

AirSim provides an interface to define vehicles as a rigid body where the actuators are modeled as sources of forces and torques. It includes parameters such as mass, inertia, coefficients for linear and angular drag, coefficients of friction and restitution etc. and are used by the physics engine to compute rigid body dynamics.

Formally, a vehicle is defined as a collection of K vertices placed at positions $\{\mathbf{r}_1,..,\mathbf{r}_k\}$ and normals $\{\mathbf{n}_1,..,\mathbf{n}_k\}$, each of which experience a control input $\{u_1,..,u_k\}$. The forces and torques from vertices are always assumed to be generated in the same direction as their corresponding 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 [16]:

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

Here C_T and C_{pow} are the thrust and the power coefficients respectively and are based on the physical characteristics of the propeller, σ 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 compute 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 [26]. 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 [12] 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. 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:

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

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:

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

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

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

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. For Pressure, we use 1976 U.S. Standard Atmosphere model [25] for altitude below 51 kilometers and switch to the model in [3] beyond that up to 86 km. Then, the air density at standard pressure P and temperature T is $\sigma = \frac{P}{R \cdot T}$ (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 take the kinematic state for each body along with forces and torques and compute the next kinematic state. We strive for an efficient physics engine that can provide a high frame rate (1000 Hz). A higher frame rate is desirable for enabling real-time simulation applications, for example, human operated flights assisted by software-controlled features such as automatic emergency maneuvers. 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 to simulate vehicles as rigid bodies.

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 [27]:

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

Here C_{lin} is the linear air drag coefficient, A is the vehicle cross-section and σ 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 ω. There are two types of drag torques generated due to angular motion experienced by ds: (1) torque produced by the tangential component of the velocity i.e. friction torque (2) torque produced by the normal component of the velocity i.e. shear stress. Our implementation focuses on friction torque as it is the dominating term. As the linear velocity $\mathbf{dv}$ experienced by ds is given by $\mathbf{r}_{ds} \times \omega$, we can now use the linear drag equation for ds [22]:

$$|\mathbf{dF}| = \frac{1}{2}\sigma|\mathbf{r}_{ds} \times \omega|^2 C_{lin} ds, \qquad \text{where direction of } \mathbf{dF} \text{ is } -\mathbf{r}_{ds} \times \omega.$$

Now, the drag torque is computed by integrating over the entire surface: $\tau_d = \int_S \mathbf{r}_{ds} \times \mathbf{dF}$. To simplify the implementation, we approximate the body of the vehicle as a 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 τ_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:

$$\mathbf{F}_{net} = \sum_i \mathbf{F}_i + \mathbf{F}_d \qquad \text{and} \qquad \tau_{net} = \sum_i [\tau_i + \mathbf{r}_i \times \mathbf{F}_i] + \tau_d$$

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

3.3.3 Integration

We update the position $\mathbf{p}_{k+1}$ of the body at time $k+1$ by integrating the velocity from the previous time step k. In our implementation we use Verlet integration instead of Runge Kutta for stability and simplicity while being nearly as efficient as Euler integration (see [11] for details). Formally,

$$\mathbf{v}_{k+1} = \mathbf{v}_k + dt \cdot \mathbf{a}_k/2 \qquad\qquad \mathbf{p}_{k+1} = \mathbf{p}_k + dt \cdot \mathbf{v}_{k+1}$$

The angular velocity is updated in similar manner: $\omega_{k+1} = \omega_k + dt \cdot \alpha_k/2$. However updating orientation isn't straight forward. One of the exisitng approaches maintains the orientation as a rotation matrix that is updated every time step. However this results in a slow drift which must be fixed at regular intervals using expensive computations. Alternative approach is to maintain rotations as quaternion which is much more efficient, numerically stable and trivially normalizable. One of the problem, however, is that the quaternion is in the world frame while the angular velocity is in the body's frame. Our approach uses this formulation.

The central step of our approach is to 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 $\alpha_{dt} = |\omega| \cdot dt$ and axis by $u = \omega/|\omega|$. This allows us to compute equivalent change in quaternion $\mathbf{q}_{dt}$ representing the change in orientation in time dt. Note that $\mathbf{q}_{dt}$ is in body's reference 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 $\mathbf{q}_{k+1}$. We can this via: $q_{k+1} = q_k \cdot q_{dt}$.

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.

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. Like other components, sensor models are expressed as a hierarchy of 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 [24] for more realistic behavior in long flights. Formally, if we denote the current bias factor as b_k then the drift is modeled as:

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

Here t, 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 $s = 0.5mbar$ based on data available in

[10] as well as experiments we conducted. This pressure p is then added with white noise drawn from zero mean Gaussian distribution with standard deviation set from datasheet (for example, 0.24 for the MEAS MS56112 sensor). Finally we convert the pressure to altitude using barometric formula used by the sensor's driver:

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

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) [28]. We model these by adding white noise and bias drift over time to the ground truth. For gyroscope, given the true angular velocity 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 many times specified directly in datasheets. Accelerometer output is computed in the similar manner.

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) [17] as our rendering platform. UE4 offers several features that made it an attractive choice including the fact that it is open source and usable for applications on Linux, Windows and OSX. UE4 brings some of the cutting edge graphics features such as physically based materials, photometric lights, planar reflections, ray traced distance field shadows and etc. Figure 1 shows a screen-shot from AirSim which highlight the near to real-world rendering quality. Further, there is a large marketplace [8] where various pre-made elaborate environments are available and created using large scale photogrammetry.

4 Experiments

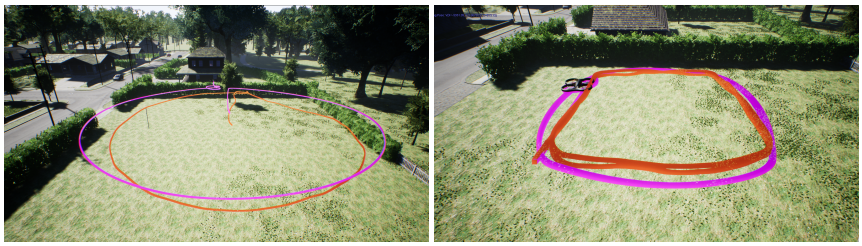
We perform experiments primarily to evaluate how close the flight characteristic of a real quadrotor flying in real-world is to that of a simulation of the same vehicle in AirSim. We also evaluate some of our sensor models against the real-world sensors.

Hardware Platform: Experiments were performed using a Pixhawk v2 hardware. Real-world flights were performed with the Pixhawk mounted on a Flamewheel quadrotor frame, together with a Gigabyte 5500 Brix running Ubuntu 16.04. For the real-world experiments, the sensor measurements were recorded on that Pixhawk itself. We instantiated the same quadrotor in AirSim and used the simulated sensor models. The AirSim MavLinkTest library was used to perform repeatable offboard control for both the real-world and the simulated flights.

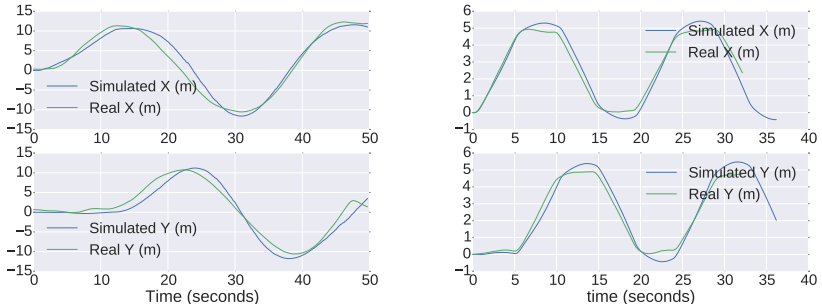
Trajectory Evaluation: We fly the quadrotor in the simulator in two different patterns: (1) trajectory in square shape with each side being 5m long (2) trajectory in circle shape with radius being 10m long. We then use exact same commands to fly the real vehicle. For both the simulation and the real-world 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 been caused by various factors such as integration errors, vehicle model approximations, mild random winds etc.



(a) Circle maneuver (b) Square maneuver

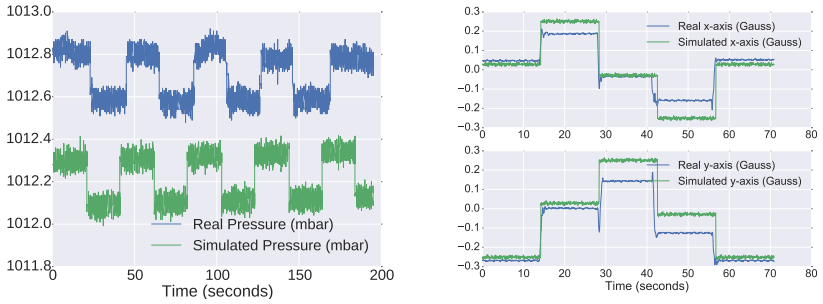


(c) Space-Time Plot for Circle (d) Space-Time Plot for Square

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.

Sensor Models: Besides evaluating the entire simulation pipeline we also investigated individual component models, namely the barometer (MEAS MS5611-01BA [5]), the magnetometer (Honeywell HMC5883 [1]) 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. We tested these simulation models for these sensors, which we describe below:

- **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 was because 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 observe that the signals have similar characteristics. Note that the offset between the



(a) Barometer (b) Magnetometer

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

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

The goal of AirSim is to enable rapid training and development of data-driven robotic systems. In particular, the platform enables hi-fidelity simulation which in turn can be used to collect training data for building machine learning models. The core components include a physics engine with detailed models of physical phenomenon, and a high fidelity perception simulation that enables training and testing of computer vision modules. By leveraging such a simulator, we hope to effectively utilize methods such as reinforcement and imitation learning and enable simulator to real-world transfer of machine learned technologies.

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 full fledged realistic collisions. New evolving features in Unreal engine such as physics engine sub-stepping might be useful in solving this. 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. The degradation of GPS signal due to obstacles is not simulated. We plan to add this using ray tracing methods. Also the system employs simple ground detection algorithm and does not offer advanced interaction models with ground (for tires etc.) It is possible to enable these capabilities via PhysX [7]. Also, the wind is simulated via simplistic wind ve-

locity vectors. We plan to add more advanced wind effects and thermal simulations. Finally, we aim to provide additional built-in vehicles via our extensibility APIs.

References

1. 3-axis digital compass ic hmc5883l. http://goo.gl/03s9um
2. Airsim. https://github.com/Microsoft/AirSim
3. Atmospheric models. http://www.braeunig.us/space/atmmodel.htm
4. Jmavsim. https://pixhawk.org/dev/hil/jmavsim
5. Ms5611-01ba barometric pressure sensor. http://goo.gl/0rBhMU
6. Open world 2015. https://www.fxguide.com/featured/epics-unreal-engine-open-world-behind-the-scenes
7. Physx. https://developer.nvidia.com/gameworks-physx-overview
8. Unreal engine marketplace. https://www.unrealengine.com/marketplace/content-cat/assets/environments
9. Bagnell, A.: An invitation to imitation. Tech. Rep. CMU-RI-TR-15-08, Robotics Inst. (2015)
10. Burch, D.: Mariner's Pressure Atlas: Worldwide Mean Sea Level Pressures and Standard Deviations for Weather Analysis and Tropical Storm Forecasting. Starpath School of Nav. (2014)
11. Butcher, J.C.: Numerical methods for ordinary differential equations (2016)
12. Chulliat, A.S., Macmillan, P., Alken, C., Beggan, M., Nair, B., Hamilton, A., Woods, V., Ridley, S., Maus, Thomson, A.: The us/uk world magnetic model for 2015-2020: Technical report. NOAA National Geophysical Data Center (2015)
13. Furrer, F., Burri, M., Achtelik, M., Siegwart, R.: Robot Operating System (ROS): The Complete Reference, chap. RotorS—A Modular Gazebo MAV Simulator Framework (2016)
14. Gao, Y., Peters, J., Tsourdos, A., Zhifei, S., Meng Joo, E.: A survey of inverse reinforcement learning techniques. International Journal of Intelligent Computing and Cybernetics (2012)
15. Jackson, J., Ellingson, G., McLain, T.: Rosflight: A lightweight, inexpensive mav research and development tool. In: ICUAS (2016)
16. J.B. Brandt R.W. Deters, G.A., Selig, M.: Uiuc propeller database, university of illinois at urbana-champaign. http://m-selig.ae.illinois.edu/props/propDB.html (2017)
17. Karis, B., Games, E.: Real shading in unreal engine 4. In: Proc. Physically Based Shading Theory Practice (2013)
18. Kober, J., Bagnell, J.A., Peters, J.: Reinforcement learning in robotics: A survey. The International Journal of Robotics Research (2013)
19. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IROS (2004)
20. Meier, L., Tanskanen, P., Fraundorfer, F., Pollefeys, M.: Pixhawk: A system for autonomous flight using onboard computer vision. In: ICRA (2011)
21. Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., Von Stryk, O.: Comprehensive simulation of quadrotor uavs using ros and gazebo. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots (2012)
22. Nakayama, Y., Boucher, R.: Introduction to fluid mechanics (1998)
23. Pan, S.J., Yang, Q.: A survey on transfer learning. Tran. on Knowledge and Data Eng. (2010)
24. Sabatini, A.M., Genovese, V.: A stochastic approach to noise modeling for barometric altimeters. Sensors (Basel, Switzerland) **13**(11) (2013)
25. Stull, R.: Practical Meteorology: An Algebra-based Survey of Atmospheric Science. University of British Columbia (2015)
26. Tapley, B., Ries, J., Bettadpur, S., Chambers, D., Cheng, M., Condi, F., Poole, S.: The ggm03 mean earth gravity model from grace. In: AGU Fall Meeting Abstracts (2007)
27. Taylor, J.R.: Classical mechanics. University Science Books (2005)
28. Woodman, O.J.: An introduction to inertial navigation. University of Cambridge (2007)