RotorPy: A Python-based Multirotor Simulator with Aerodynamics for Education and Research

Spencer Folk, James Paulos, and Vijay Kumar

***Abstract*— Simulators play a critical role in aerial robotics both in and out of the classroom. We present *RotorPy*, a simulation environment written entirely in Python intentionally designed to be a lightweight and accessible tool for robotics students and researchers alike to probe concepts in estimation, planning, and control for aerial robots. *RotorPy* simulates the 6-DoF dynamics of a multirotor robot including aerodynamic wrenches, obstacles, actuator dynamics and saturation, realistic sensors, and wind models. This work describes the modeling choices for *RotorPy*, benchmark testing against real data, and a case study using the simulator to design and evaluate a model-based wind estimator.**

I. INTRODUCTION

Dynamics simulation environments aid robotics education and research, providing a playground for rapid experimentation and evaluation of robotic design, perception, and action. Aerial robots, or UAVs, are a complicated application domain–unstable dynamics requiring high speed sensors, actuators, controllers, and planners, and complex aerodynamic interactions with the environment and other UAVs–placing added demands on simulation tools necessary for synthesis and analysis. Existing simulators, driven by target applications, tend to prioritize compute speed with hardware integration like *RotorS* [1] and *Agilicious* [2], or photorealistic visualization like with *Airsim* [3] and *Flightmare* [4]. Also, reinforcement learning (RL) for UAVs is sprouting simulation environments like *Gym-PyBullet-Drones* [5] purpose-built for use with common Python-based RL toolkits. The trend seems to be towards increasingly complex and elaborate codebases requiring a high level of expertise to navigate and understand their modeling choices, making it hard to decide whether or not a simulator will fit the needs of a new user. To that end, we developed a new simulation environment called *RotorPy*[1], which prioritizes accessibility, transparency, and educational value, serving as a tool for learning and exploration in aerial robotics both for students and researchers. Initially created as a teaching aid for a robotics course at the University of Pennsylvania, *RotorPy* was designed to be lightweight, easy to install, and accessible to engineers with working knowledge of Python.

This paper introduces *RotorPy*'s modeling choices, structure, and features contributing to an effective environment for probing aspects of UAVs; and then, we present a case study using the simulator to design a model-based wind estimator.

S. Folk and V. Kumar are with the GRASP Laboratory, University of Pennsylvania, Philadelphia, PA, USA, {sfolk, kumar}@seas.upenn.edu. J. Paulos is with Treeswift, Philadelphia, PA, USA, jpaulos@gmail.com. This work is not related to Treeswift.

[1]github.com/spencerfolk/rotorpy

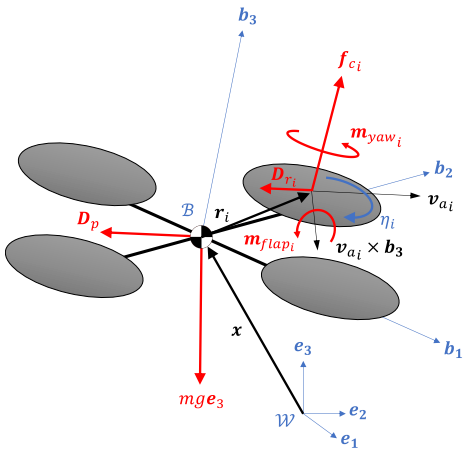


Fig. 1: Free body diagram of a UAV subject to control and aerodynamic wrenches. Relative airflow through the fluid medium, $\boldsymbol{v}_a$, produces additional wrenches in the form of aerodynamic drag on the frame, $\boldsymbol{D}_p$ and rotors, $\boldsymbol{D}_{r_i}$.

II. MODELING

RotorPy includes a quadrotor UAV model with aerodynamic wrenches, inertial and motion capture sensors, cuboid obstacle environments, and spatio-temporal wind fields.

A. Multirotor dynamics

Following Figure 1, we model a multirotor UAV with coplanar rotors using the Newton-Euler equations:

$$\dot{\boldsymbol{x}} = \boldsymbol{v} \tag{1}$$

$$\dot{\boldsymbol{v}} = \frac{1}{m} R(\boldsymbol{f}_c + \boldsymbol{f}_a) - g\boldsymbol{e}_3 \tag{2}$$

$$\dot{R} = R\hat{\boldsymbol{\Omega}} \tag{3}$$

$$\dot{\boldsymbol{\Omega}} = J^{-1}(\boldsymbol{m}_c + \boldsymbol{m}_a - \boldsymbol{\Omega} \times J\boldsymbol{\Omega}) \tag{4}$$

where $\boldsymbol{x} \in \mathbb{R}^3$ and $\boldsymbol{v} \in \mathbb{R}^3$ are respectively the position and velocity vectors; $R \in SO(3)$ is the rotation from the body frame to the world frame; $\boldsymbol{\Omega} \in \mathbb{R}^3$ is the angular velocity; m is the total mass and J is the inertia tensor expressed in the body frame.

The terms $\boldsymbol{f}_c \in \mathbb{R}^3$ and $\boldsymbol{m}_c \in \mathbb{R}^3$ constitute the control wrench, i.e., the forces and torques produced by the rotor thrust and drag torque. We model the control wrench with

$$\boldsymbol{f}_c = k_\eta \sum_{i=1}^{n} \eta_i^2 \boldsymbol{b}_3 \tag{5}$$

$$\boldsymbol{m}_c = k_m \sum_{i=1}^{n} \epsilon_i \eta_i^2 \boldsymbol{b}_3 + \sum_{i=1}^{n} \boldsymbol{r}_i \times \boldsymbol{f}_{c_i}. \tag{6}$$

Here, η_i is the i'th rotor speed, $\boldsymbol{r}_i$ is the vector from the center of mass to the rotor hub, and $\epsilon_i \in \{-1, 1\}$ is the rotor's direction of rotation. We assume that each rotor has the same static thrust, k_η, and drag torque, k_m, coefficients which can be identified using thrust stand tests.

B. Aerodynamic wrenches

Together, the vectors $\boldsymbol{f}_a \in \mathbb{R}^3$ and $\boldsymbol{m}_a \in \mathbb{R}^3$ make up the aerodynamic wrench, which is a collection of forces and torques produced by the relative motion of the UAV through a fluid medium. There are multiple physical phenomena that produce $\boldsymbol{f}_a$ and $\boldsymbol{m}_a$ on the UAV (see [6] and references therein). These effects are all dependent on the relative body airspeed, $\boldsymbol{v}_a = R^\top(\dot{\boldsymbol{x}} - \boldsymbol{w})$ where $\boldsymbol{w} \in \mathbb{R}^3$ is a local wind vector in the world frame. We lump these effects into three contributions to the aerodynamic wrench: parasitic drag, rotor drag, and blade flapping.

1) Parasitic drag: Parasitic drag is the combination of skin friction and pressure drag acting on the body of the UAV. It is characteristically proportional to the airspeed squared:

$$\boldsymbol{D}_p = -C||\boldsymbol{v}_a||_2 \boldsymbol{v}_a \tag{7}$$

where $C = \text{diag}(c_{Dx}, c_{Dy}, c_{Dz})$ is a matrix of parasitic drag coefficients corresponding to each body axis.

2) Rotor drag: In contrast to parasitic drag, which is dominant at higher airspeeds, rotor drag can have a surprisingly large presence on small UAVs even at lower airspeeds. The physical phenomenon responsible for rotor drag is the dissymmetry of lift produced by a rotor in forward flight, whereby the advancing blade experiences a higher airspeed than the retreating blade producing an imbalance of forces on the rotor. We adopt the rotor drag model used in [6] in which the drag force is proportional to the product of the airspeed and the rotor speed.

$$\boldsymbol{D}_{r_i} = -K\eta_i \boldsymbol{v}_{a_i} \tag{8}$$

where $K = \text{diag}(k_d, k_d, k_z)$ is a matrix of rotor drag coefficients[2] corresponding to each body axis.

3) Blade flapping: Dissymmetry of lift at the advancing and retreating sides of the rotor will also cause the rotor blades to deflect up and down as they revolve in a flapping motion. Svacha *et al.* [6] provides experimental evidence for flapping moments even for small UAVs with rigid rotors. This is a very complex phenomenon that can produce both longitudinal and lateral moments depending on the rigidity of the blades [7]. Our model expresses blade flapping as a longitudinal moment following [6]

$$\boldsymbol{m}_{flap_i} = -k_{flap}\eta_i \boldsymbol{v}_{a_i} \times \boldsymbol{b}_3 \tag{9}$$

with k_{flap} being the flapping coefficient.

The total aerodynamic force in the body frame is $\boldsymbol{f}_a = \boldsymbol{D}_p + \sum_{i=1}^{n} \boldsymbol{D}_{r_i}$, and the total moment is $\boldsymbol{m}_a = \sum_{i=1}^{n} (\boldsymbol{m}_{flap_i} + \boldsymbol{r}_i \times \boldsymbol{D}_{r_i})$.

[2]As noted in [6], the k_z term isn't actually a source of drag, but rather a linear approximation of loss of thrust due to change in inflow. However, it resembles an effective drag on the body z axis.

C. Actuator dynamics

Even for very small UAVs, the motors take time to settle to a commanded speed. Capturing this effect has proven to be important especially for RL applications [5]. We model the actuator delay using a first order process:

$$\dot{\boldsymbol{\eta}} = \frac{1}{\tau_m}(\boldsymbol{\eta}_c - \boldsymbol{\eta}) \tag{10}$$

where $\boldsymbol{\eta}_c \in \mathbb{R}^n$ are the commanded rotor speeds and τ_m is the motor time constant–it can be identified using static thrust stand testing.

D. Sensors

1) Inertial measurement unit: The simulator's inertial measurement unit (IMU) measurement is given by:

$$\boldsymbol{h}_{IMU} = \begin{bmatrix} R_{\mathcal{I}}^{\mathcal{B}}(\dot{\boldsymbol{v}} + g\boldsymbol{e}_3) + \boldsymbol{a}_{IMU} + \boldsymbol{b}_a + \boldsymbol{\nu}_a \\ \boldsymbol{\Omega} + \boldsymbol{b}_g + \boldsymbol{\nu}_g \end{bmatrix} \tag{11}$$

where $\dot{\boldsymbol{v}}$ is given by equation 2, $\boldsymbol{a}_{IMU} = \boldsymbol{\Omega} \times (\boldsymbol{\Omega} \times \boldsymbol{r}_{IMU})$, $\boldsymbol{r}_{IMU}$ and $R_{\mathcal{I}}^{\mathcal{B}}$ are the position and orientation of the sensor in the body frame, and $\boldsymbol{\nu}_{(\cdot)} \sim \mathcal{N}(0, \Sigma_{(\cdot)})$ are sensor noises. The biases $\boldsymbol{b}_{(\cdot)}$ are driven by random walk to simulate drift.

2) External motion capture: The external motion capture sensor provides information about the pose and twist of the robot in the world frame.

$$\boldsymbol{h}_{MC} = \begin{bmatrix} \boldsymbol{x} + \boldsymbol{\nu}_x \\ \boldsymbol{v} + \boldsymbol{\nu}_v \\ \boldsymbol{q} \oplus \boldsymbol{q}_\nu \\ \boldsymbol{\Omega} + \boldsymbol{\nu}_\Omega \end{bmatrix}. \tag{12}$$

Above, $\boldsymbol{q}$ is the quaternion representation of R, $\oplus$ is the quaternion group operation, and $\boldsymbol{q}_\nu$ is a quaternion formed by small noise perturbations following [8].

E. Wind

Wind is modeled by treating the local average wind acting on the center of mass as an additional state vector. How this state evolves depends on the chosen wind profile. *RotorPy* offers flexibility by supporting both spatial and temporal wind profiles. Several profiles like step changes, sinusoids, and the Dryden wind turbulence model are included for evaluating controller robustness or estimation accuracy.

III. SIMULATION FRAMEWORK

RotorPy is written entirely in Python–a deliberate choice originally made to serve instructional needs. While this choice might come at a performance cost, the readability and widespread use of Python in scientific computing is the key to this simulator's accessibility and low barrier to entry which is beneficial for education, as originally intended, but also for research. Python also makes installation of both *RotorPy* and its dependencies possible with one command.

A. Usage

RotorPy is a collection of modules that can be imported to scripts anywhere. The `Environment` class makes it possible to create, run, and analyze multiple simulations, all with potentially unique configurations, in just a single Python file. We believe this design principle makes *RotorPy* stand out among other simulators which typically run in a self-contained manner. In contrast, our simulator is purposefully exportable in a manner conducive to studies that require lots of data (e.g., reinforcement learning, design parameter search, controller verification).

The environment needs a vehicle, controller, and planner; examples of these are all provided out of the box.

```
sim_instance = Environment(vehicle, controller,
    trajectory, *args)
```

The `Environment` also has options to add wind and obstacles, configure sensor intrinsics and extrinsics, and more.

Running the simulator only takes one line:

```
results = sim_instance.run(duration, *args)
```

The output of `run()` is a dictionary containing the ground truth states, desired state from the trajectory planner, sensor measurements, and controller commands. In addition to optional auto-generated plots and simple animations for quick user assessment, we provide a script for automatically converting the results into a *Pandas* DataFrame for larger scale data analysis.

B. Numerical integration

UAVs are hybrid systems–the dynamics are continuous but control occurs in discrete instances–motivating an approach to numerical integration that preserves the continuity of dynamics in between controller updates. To that end, *RotorPy* uses an RK45 integrator with variable step size[3]. An added benefit of the variable step size is that we can run simulations with larger time steps, reducing compute cost, while preserving the integration accuracy.

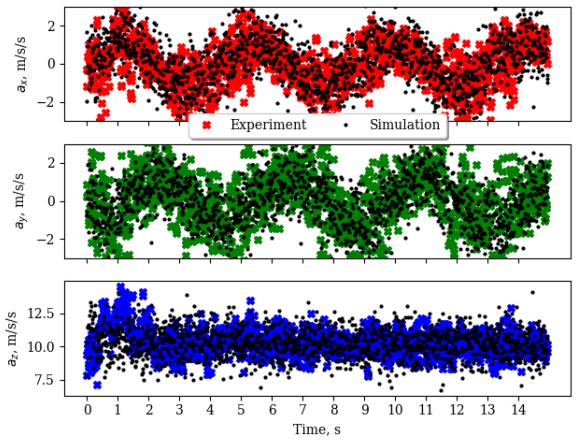
IV. BENCHMARKING

In order to verify our models, especially for the sensors, we collected flight data from a Crazyflie 2.1[4] performing a series of aggressive maneuvers. In this paper, we present one instance of our hardware trials in which the Crazyflie is commanded to fly in a tight circle at speeds up to 2.5 m/s.

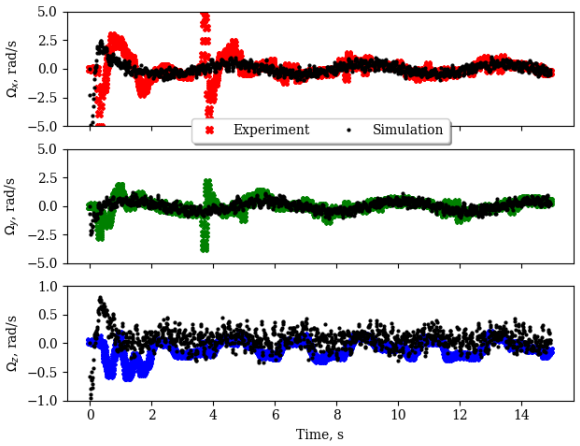
A. Hardware setup

A motion capture system sends pose and twist data at 100Hz to a base station computer, which uses a nonlinear geometric controller to generate a command based on the current state and desired trajectory. *RotorPy*'s controller and trajectory generator are designed to be compatible with our lab hardware. The Crazyflie uses onboard PID controllers to track the collective thrust and attitude commands from the simulator's controller.

[3]docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html
[4]www.bitcraze.io/



(a) Linear accelerometer measurements in the body frame.



(b) Angular velocity measurements in the body frame.

Fig. 2: A comparison between the simulated and actual measurements from the Crazyflie's IMU while tracking a 1.5 m circle at 2.5 m/s.

B. Circle comparison

In Figure 2, we compare the IMU measurements from simulation and the Crazyflie. This comparison principally highlights the simulator's sensor and aerodynamic models. For our co-planar configuration, the accelerations in the x and y body axes isolate the drag forces for comparison, since we would otherwise expect zero accelerations in the absence of the aerodynamic wrenches [9].

V. CASE STUDY: WIND ESTIMATION

We demonstrate the utility of *RotorPy* by evaluating a custom Bayesian filter for estimating the local wind vector, $\boldsymbol{w}$, using measurements from the navigation system rather than a dedicated wind sensor–this approach can be classified as indirect wind estimation [10]. The estimator is implemented in simulation as an unscented Kalman Filter (UKF), using the simulator's accelerometer and the motion capture sensors to observe the wind vector. The process model makes several simplifications: linearized attitude dynamics and a version of

the aerodynamics that only considers parasitic drag. For each evaluation, a calibration procedure collects simulated flight data of the quadrotor with randomized parameters, and then fits quadratic drag coefficients for the process model.

Figure 3 summarizes the average RMSE over 50 evaluations of the filter using randomized quadrotor parameters in Table I. In half of the trials, the filter's RMSE falls around or under 0.5 m/s; however, performance is poor in cases where the calibration procedure fails to find good drag coefficients, like when the real drag coefficients are small. Figure 4 is one instance of the evaluation, comparing the actual wind components to that estimated from the filter. This trial highlights an important model discrepancy: the process model uses the commanded thrust, not the actual thrust, produced by the rotors. In cases of overwhelming winds like in Figure 4, the motors are saturated which causes a model discrepancy between the commanded and actual thrust, leading to estimation error.

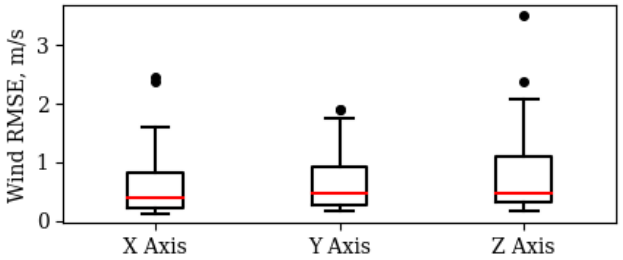


Fig. 3: Monte Carlo evaluation of the wind filter over 50 simulations; each instance has randomized mass, drag coefficients, and average wind magnitudes.

TABLE I: Randomized quadrotor parameters for the Monte Carlo evaluation of the wind filter. Symbols are consistent with Section II.

Parameter	Unit	Range (min-max)
m	kg	0.375–0.9375
c_{Dx}	N-(m/s)$^{-2}$	0–1(10^{-3})
c_{Dy}	N-(m/s)$^{-2}$	0–1(10^{-3})
c_{Dz}	N-(m/s)$^{-2}$	0–2(10^{-2})
k_d	N-rad-m-s^{-2}	0–1.19(10^{-3})
k_z	N-rad-m-s^{-2}	0–2.32(10^{-3})

VI. SUMMARY

This work presents *RotorPy*, a UAV simulation environment that is designed to be accessible to engineers with working knowledge of Python. The simulator is packaged with a 6-DoF model of a quadrotor UAV with aerodynamics and motor dynamics, realistic sensors, obstacles, and wind models. In addition, we provide a tracking controller, multiple trajectory generation methods, and a wind estimation filter for convenience. For verification, we compare our simulator to real data collected from a Crazyflie performing aggressive trajectories that highlight the aerodynamic forces present in high speed flight. We believe that *RotorPy* can be a useful tool both in and out of the classroom as a way to dig deep into concepts in estimation, planning, and control for UAVs in the presence of high winds. This is demonstrated in our case study, which looks at using *RotorPy* to evaluate a model-based wind estimator. Future developments include broader support for different UAV archetypes and incorporation of a fast fluid dynamics solver for native spatio-temporal wind field generation.

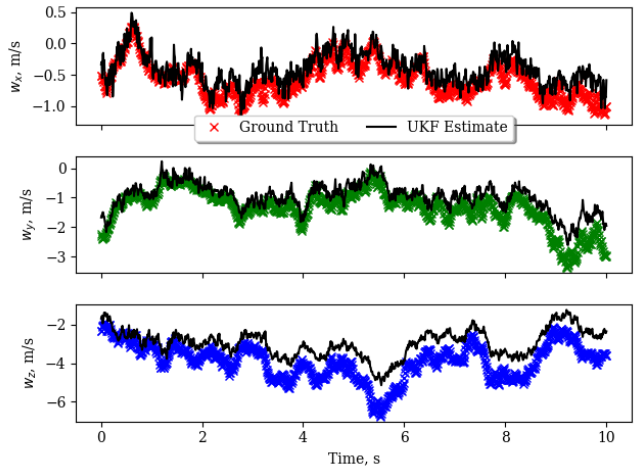


Fig. 4: A simulated instance of the unscented Kalman Filter estimating the local wind velocity vector for a quadrotor subject to Dryden wind gusts.

REFERENCES

[1] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Cham: Springer International Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9_23

[2] P. Foehn, E. Kaufmann, A. Romero, R. Penicka, S. Sun, L. Bauersfeld, T. Laengle, G. Cioffi, Y. Song, A. Loquercio, and D. Scaramuzza, "Agilicious: Open-source and open-hardware agile quadrotor for vision-based flight," *AAAS Science Robotics*, 2022.

[3] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[4] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Conference on Robot Learning*, 2020.

[5] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 7512–7519.

[6] J. Svacha, J. Paulos, G. Loianno, and V. Kumar, "Imu-based inertia estimation for a quadrotor using newton-euler dynamics," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 3861–3867, 2020.

[7] R. W. Allen, "Flapping characteristics of rigid rotor blades," *Journal of the Aeronautical Sciences*, vol. 13, no. 4, pp. 183–186, 1946. [Online]. Available: https://doi.org/10.2514/8.11343

[8] J. Sola, "Quaternion kinematics for the error-state kalman filter," *arXiv preprint arXiv:1711.02508*, 2017.

[9] P. Martin and E. Salaün, "The true role of accelerometer feedback in quadrotor control," in *2010 IEEE International Conference on Robotics and Automation*, 2010, pp. 1623–1629.

[10] P. Abichandani, D. Lobo, G. Ford, D. Bucci, and M. Kam, "Wind measurement and simulation techniques in multi-rotor small unmanned aerial vehicles," *IEEE Access*, vol. 8, pp. 54 910–54 927, 2020.