J Intell Robot Syst (2010) 57:407–415
DOI 10.1007/s10846-009-9354-6

UAS Flight Simulation with Hardware-in-the-loop Testing and Vision Generation

Jeffery Saunders · Randal Beard

Received: 1 February 2009 / Accepted: 1 August 2009 / Published online: 27 August 2009

Abstract UAS flight simulation for research and development is a difficult problem because each airframe requires accurate physical models, control systems, an organized method of testing new control systems, virtual cameras for vision-based control, and methods of testing new control in the transition from simulation to flight tests. In an environment where researchers are temporary, such as a university, a standard research and development platform with these properties expedites prototyping and prevents code loss when an employee leaves. We develop a research simulation which conforms to all of these properties inside a Matlab environment. A series of mex functions provide connections to the autopilot for hardware-in-the-loop testing, graphical interfaces, and vision processing. The option to write C mex functions offers a seamless method of porting code to embedded systems, minimizing coding errors. We demonstrate fast prototyping by showing flight test data where the simulation provided virtual vision data to avoid virtual obstacles.

Keywords Flight simulation · Micro air vehicle · Unmanned air system · Matlab · Hardware in the loop

1 Introduction

Unmanned Air Systems (UASs) take on tasks to replace piloted aircraft in situations where human intervention is dangerous, too expensive, or infeasible. In military tasks, they remove danger to human pilots by flying surveillance, reconnaissance,

J. Saunders · R. Beard (✉)
Electrical and Computer Engineering,
Brigham Young University, Provo, UT 84602, USA
e-mail: beard@byu.edu

J. Saunders
e-mail: saunders.jeff@gmail.com

and attack missions without human involvement. Domestic missions include forest fire monitoring, search and rescue, and surveillance. Very often, small, relatively inexpensive UASs accomplish these tasks. UASs not only remove human risk, but lower the overall costs involved in mission tasks.

The high number of potential applications and the low cost of small UASs have kindled research and development in small autopilots. The decrease in size of the required sensors and integrated circuits have allowed the development of light-weight autopilots capable of navigating airframes with wingspans as small as 6 inches [1, 2]. The sensors available for large aircraft are not feasible on small UASs because of payload limitations. They require small sensors that are typically less accurate. Small UASs typically have fast dynamics, making accurate sensors even more important. While the technology of sensors improve, inner loop autopilot control continues to develop using alternative sensors such as computer vision [4]. Our objective is to create a simulation platform that facilitates development of these technologies.

The purpose of this paper is to describe a simulation environment that expedites prototyping by providing mechanisms to test new guidance, navigation, and control algorithms on various types of UASs, including vision-based control. The base simulation is simple, but offers enough features to aid in development of complex algorithms.

The University of Florida developed a simulation testbed for rapid vision processing [3]. Their goal was to accelerate the process of development of vision-based control inside simulation and smoothly transition the control to flight tests. However, their system required extensive hardware resources. Our goals are similar, but to do so on a single laptop computer. The objective is to create a simulation environment where vision-based control can be rapidly developed, tested with hardware in the loop, and rapidly transitioned to flight tests. We briefly discuss an application in which this was accomplished and describe the associated flight [6].

There are five requirements for the simulator:

1. Modularity to support different airframes and research,
2. Rapid prototyping,
3. Hardware-in-the-loop testing,
4. Graphical interface with camera simulation for vision processing, and
5. Code standardization to prevent code loss when employees leave.

Each of these requirements are discussed in the paper. Section 2 presents the structure of the simulation code. Section 3 discusses the physical subsystem and gives an example of a fixed-wing airframe. The guidance and navigation systems, including hardware-in-the-loop testing, is discussed in Section 4 followed by the graphical interface in Section 5. An example flight test in which we used hardware-in-the-loop vision processing is shown in Section 6 with a conclusion in Section 7.

2 Overview

A requirement of the simulator is modularity. At Brigham Young University, we have a wide variety of airframes, including fixed wing aircraft, quadrotors,

helicopters, and vertical take-off and land (VTOL) aircraft. We want to incorporate each of these different airframes into the simulator with minimal change, including physics and autopilot capabilities. The simulator is encoded in Matlab's Simulink.

Simulink has many beneficial characteristics for a simulator using multiple airframes and autopilots including:

1. A scripting interface for fast development,
2. A large set of engineering script libraries including control and computer vision,
3. A customizable library for simulation blocks that can be used for different airframes,
4. Simple data plotting functions,
5. An interface for C++ code (mex functions).

The block diagram of the simulation structure is shown in Fig. 1, which shows three subsystems. The first subsystem is the Guidance and Navigation System which includes the State Estimator, the High Level Path Planner, the Autopilot, and the Vision Processor required.

The second subsystem is the Physical System which includes the sensors onboard the UAS and the physics of the airframe. This subsystem is important when changing airframes. A fixed-wing aircraft has different flight characteristics than a quadrotor. To make this simulation universal, we store the physics engine of each airframe in the Simulink library. Each physics engine can be dragged from the Simulink library to the simulation.

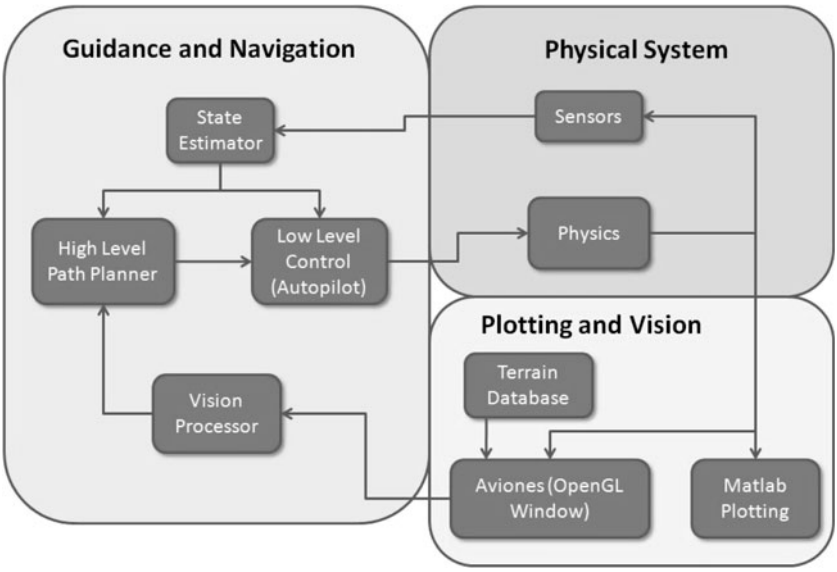

Fig. 1 The simulation is subdivided into three subsystems. Each subsystem contains Simulink blocks that can be interchanged for different autopilots, airframes, etc. as required

The third subsystem is Plotting and Vision which has two purposes. The first purpose is to display graphs, plots, and data as required. The second purpose is the graphical display called Aviones. Aviones displays the UAS in a terrain and simulates a camera for computer vision.

Our end goal is not to produce a good implementation in simulation, but to produce flight results from the theory using code developed in simulation. Many embedded systems are coded using C/C++. To implement an autopilot on a microprocessor, the Matlab script must be converted to C code. The conversion to C code also opens possibilities for bugs and problems that were not present in the script version. To aid in the transition, Matlab has a C/C++ interface called mex functions. The script can be written in C/C++ code and tested in the simulation environment without changing any other details. This limits the programming bugs and errors present in the embedded system.

In an educational environment, employment length is relatively short. In that time, employees implement their research on the current autopilot platform, or in a specialized simulation. The code is rarely reusable and often poorly documented. Code loss is prevented in a standard simulation platform and it provides a means of documentation. Each employee provides a section of code in a Simulink block that corresponds to their research, ideally using a script implementation and a mex function implementation. The state variables are identical for each code implementation, allowing each block to be added to other copies of the simulation. Each Simulink block includes a section for documentation. When a code implementation is required for another project, we use the Simulink block previously created. This standardized method minimizes code loss.

3 Physics

The Physics section simulates the physical system of the airframe. Matlab plays a key role in helping a researcher write and debug the code of the physical system. In the case of a fixed-wing aircraft, there are typically 12 vehicle states and 2 camera states which the simulator uses as a standard state vector. They are inertial position (p_n, p_n, p_d), body frame velocity (u, v, w), roll ϕ, pitch θ, yaw ψ, angular accelerations (p, q, r), and the azimuth α_a and elevation α_{el} of the camera. The force equations we use are

$$\begin{pmatrix} f_x \\ f_y \\ f_x \end{pmatrix} = mg \begin{pmatrix} -\sin\theta \\ \cos\theta \sin\phi \\ \cos\theta \cos\phi \end{pmatrix} + \frac{1}{2}\rho V_a^2 S \begin{pmatrix} C_X(\mathbf{x}, \boldsymbol{\delta}) \\ C_Y(\mathbf{x}, \boldsymbol{\delta}) \\ C_Z(\mathbf{x}, \boldsymbol{\delta}) \end{pmatrix}, \tag{1}$$

$$\begin{pmatrix} l \\ m \\ n \end{pmatrix} = \frac{1}{2}\rho V_a^2 S \begin{pmatrix} \frac{b}{2} C_X(\mathbf{x}, \boldsymbol{\delta}) \\ \bar{c} C_Y(\mathbf{x}, \boldsymbol{\delta}) \\ \frac{b}{2} C_Z(\mathbf{x}, \boldsymbol{\delta}) \end{pmatrix}, \tag{2}$$

where C_* are functions to determine effective lift, $\mathbf{x}$ is the states, ρ is the fluid density, S is the incidence area of the wing, and $\boldsymbol{\delta}$ is the surface deflections. The differential equations for a fixed-wing aircraft using those forces are

$$\begin{pmatrix} \dot{p}_n \\ \dot{p}_e \\ \dot{p}_h \end{pmatrix} = \begin{pmatrix} c\theta c\psi & s\phi s\theta s\psi - s\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\theta\psi & c\phi s\theta s\psi - s\phi c\psi \\ s\theta & -s\phi c\theta & -c\phi c\theta \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix}, \tag{3}$$

$$\begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{pmatrix} = \begin{pmatrix} rv - qw \\ pw - ru \\ qu - pv \end{pmatrix} + \begin{pmatrix} -g\sin\theta \\ g\cos\theta\sin\phi \\ g\cos\theta\cos\phi \end{pmatrix} + \frac{\rho V_a^2 S}{2m} \begin{pmatrix} C_X(\mathbf{x}, \boldsymbol{\delta}) \\ C_Y(\mathbf{x}, \boldsymbol{\delta}) \\ C_Z(\mathbf{x}, \boldsymbol{\delta}) \end{pmatrix}, \tag{4}$$

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix}, \tag{5}$$

$$\begin{pmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} = \begin{pmatrix} \Gamma_1 pq - \Gamma_2 2qr \\ \Gamma_5 pr - \Gamma_4(p^2 - r^2) \\ \Gamma_6 pq - \Gamma_1 qr \end{pmatrix} + \frac{\rho V_a^2 S}{2} \begin{pmatrix} \frac{b}{2}\left(\Gamma_3 C_l(\mathbf{x}, \boldsymbol{\delta}) + \Gamma_4 C_n(\mathbf{x}, \boldsymbol{\delta})\right) \\ \frac{\bar{c}}{J_y} C_m(\mathbf{x}, \boldsymbol{\delta}) \\ \frac{b}{2}\left(\Gamma_4 C_l(\mathbf{x}, \boldsymbol{\delta}) + \Gamma_7 C_n(\mathbf{x}, \boldsymbol{\delta})\right) \end{pmatrix}. \tag{6}$$

where Γ_* and J_y are functions of the inertia matrix, and C_n, C_ℓ, C_n are moment coefficients, g is the gravitational constant, and m is the mass of the aircraft.

We implemented these equations in Matlab script for debugging purposes, and in a mex function to lower processing time. Parameters such as lift and drag coefficients are available in a Simulink block mask and they are changeable as necessary without changing code. Each airframe has a physics block in the Simulink library. The state vector is standard for each airframe, allowing each airframe physics to be used in different simulations.

4 Guidance and Navigation

The guidance and navigation section of the simulator includes the low level control, high level control, vision processing, and sensor filters.

4.1 Autopilot

The autopilot is the low level inner loops to maintain altitude, heading, roll, pitch, heading rate, etc. Commercial autopilots commonly use a series of PID loops for low level control of an aircraft [1]. Simulink can implement an autopilot using either a Matlab script or a C++ mex function. The scripting interface offers an excellent method to implement and test new ideas. Scripting requires no compilation, allows

breakpoints, and provides complex mathematical operations with simple commands. After theoretical analysis is complete, the Matlab script can be ported to a mex function before porting it to an autopilot. The mex function provides a method of testing C code in simulation to minimize errors before moving it to an autopilot system.

4.2 High Level Path Planning

A variety of high level path planners can be developed. For example, a waypoint path planner creates waypoint paths that cause the UAV to fly a series of waypoints [5]. The high level planner provides roll, airspeed, and climb rate commands to the autopilot.

4.3 Vision Processing

Matlab has a wide variety of vision processing functions available through its scripting interface. The vision processing Simulink block can use the Matlab vision processing functions for fast development of required vision processor. The Aviones graphics block simulates a camera and sends the pixel data to the vision processing block. The pixel data from Aviones is treated as pixel data from an actual camera, even adding noise is possible. Vision processing on the data is different for each application, but the development process is simplified by script implementation. The Matlab functions offer a method to check theory validity without C-code.

For flight, a vision processing implementation in C is usually required. To ease the process of converting from Matlab script, we use a mex function with OpenCV support. OpenCV is an open source vision processing library. OpenCV has many powerful vision functions. Using OpenCV in a mex function greatly reduces the time required in converting from script to C-code. The mex function is used in the same way as a script file, allowing the C implementation to be tested in the simulation before using it in a separate program.

4.4 Hardware in the Loop

Before any flight, a control platform needs to be tested in an environment that is as close to actual flight as possible. We use the embedded autopilot with simulated physics to test the platform. In our case, the physics engine is also located on the autopilot. An RF modem transmits the telemetry data from the autopilot to the ground station. A computer program called Virtual Cockpit collects the data. A TCP/IP connection sends the telemetry data to other programs as required, including Matlab through a mex function. The simulation can use the telemetry information for any hardware-in-the-loop testing required, such as vision simulation with Aviones and vision processing with the script interface or OpenCV. Running hardware-in-the-loop using Matlab creates a vision simulation that normally isn't available for hardware-in-the-loop. This option speeds up the development cycle (Fig. 2).

The hardware-in-the-loop mechanism is also available for flight tests. While another program can be made for monitoring and vision processing to separate simulation from hardware-in-the-loop and flight tests, the option of leaving control

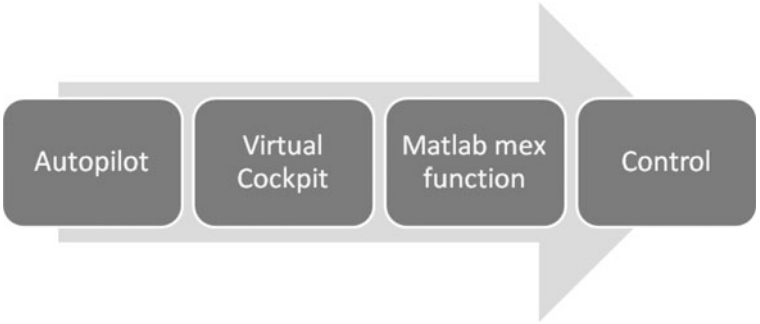

Fig. 2 Telemetry information flows from the autopilot to Virtual Cockpit via an RF modem, to Simulink via a TCP/IP connection

in Matlab speeds up development. Using Matlab for vision processing and control provides an option of using simulated vision during a flight test, increasing safety and allowing observance of flight characteristics in the control loops.

5 Graphical Interface

5.1 Matlab Plotting

Matlab has a large set of plotting functions useful for debugging, tuning, and evaluation. Debugging and plotting routines are written by employees. Many of the routines are useful and added to the Simulink library for use in other projects.

5.2 Aviones

We use another plotting interface called Aviones which was developed at BYU. Aviones is an OpenGL 3D environment that uses terrain elevation maps overlayed by satellite images of the terrain. The result is a virtual terrain matching any terrain for which elevation data is available. An example is shown in Fig. 3a. This environment is available in Matlab through a mex function. The first call to the function initializes the window with the desired size, frame rate, terrain, etc. Another call to the function updates the state of the UAS. The result is simulated video of the UAS moving through the terrain.

The viewpoint in Aviones is adjustable. There are four viewpoints available: chase, camera, satellite, and groundtrack. The chase view is a viewpoint one meter behind the UAS with the same yaw angle as the UAS. Figure 3a shows the chase view. The satellite view is simply a high altitude looking down on the terrain. The groundtrack views the UAS from the launch position. The most useful view for vision directed control is the camera view. The camera view simulates a camera at the elevation and azimuth angles given in the state vector. The pixel data from the camera view is sent to the video processing Simulink block for vision processing.

Fig. 3 Aviones generates viewpoints at various positions. The pixel data from any of them can be used for vision processing. **a** UAS chase view. **b** UAS camera view

6 Flight Tests

We conduct flight tests on a regular basis. Some of these flight tests involve our obstacle avoidance algorithms. Obstacle avoidance is dangerous to test. If an obstacle is not detected properly, the airframe may crash into the obstacle, possibly damaging the airframe and the obstacle. In addition, most obstacles are close to the ground, requiring the UAS to fly at low altitudes and relying on accurate altitude measurements to prevent it from colliding with the ground. These dangers can be alleviated by testing first using hardware-in-the-loop vision provided by the simulator described in this paper. The graphical interface section generates camera data, Matlab processes it, and sends control commands to the autopilot in flight. This allows the UAS to fly at high altitudes, in an obstacle free area, while testing obstacle avoidance routines. If performance is adequate, then a flight test with real obstacles may be warranted.

Obstacle avoidance with hardware-in-the-loop vision has been conducted many times with this simulation architecture. An example is shown in Fig. 4. In this

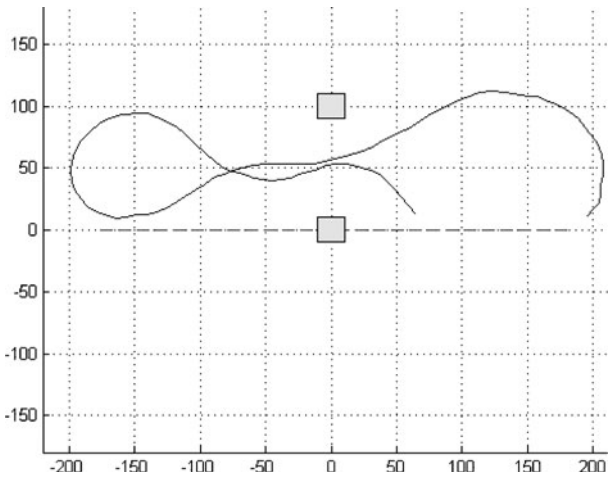

Fig. 4 The UAS avoids virtual obstacles using virtual vision with the hardware-in-the-loop simulation. The *solid line* is the UAS path, the *dashed line* is the waypoint path, and the *yellow blocks* are obstacles

example, we use the obstacle avoidance routines described in Ref [6]. As the vision processing detects the obstacles, it moves the obstacles to the edge of the camera field-of-view to avoid them. For two obstacles, each obstacle is moved to opposite sides of the camera field-of-view, thus the UAS maneuvers between the obstacles. The flight was conducted safely at an altitude of 100 meters in an obstacle free area. This flight test allowed us to monitor flight performance, tune gains, and determine if a low altitude test with real obstacles is feasible. All control code was written in Matlab script, unchanged from the purely simulated tests.

7 Conclusion

In this paper, we have described a simulator with the following properties,

1. Modularity to support different airframes and research,
2. Rapid prototyping,
3. Hardware-in-the-loop testing,
4. Graphical interface with camera simulation for vision processing,
5. Code standardization to prevent code loss when employees leave,
6. Support for flight testing with vision generation.

The simulator is a significant factor in fast development and prototyping of new research. It contributes to safe flight tests in situations where flight tests are dangerous by allowing testing at high altitudes and generated data. It also provides standardization.

References

1. Beard, R., Kingston, D., Quigley, M., Snyder, D., Christiansen, R., Johnson, W., McLain, T., Goodrich, M.A.: Autonomous vehicle technologies for small fixed-wing UAVs. J. Aerosp. Comput. Inform. Commun. **2**, 92–102 (2005)
2. Grasmeyer, J.M., Keennon, M.T.: Development of the black widow micro air vehicle. In: 39th AIAA Aerospace Sciences Meeting and Exhibit (AIAA Paper No. 2001-0127) (2001)
3. Grzywna, J.W., Jain, A., Plew, J., Nechyba, M.C.: Rapid development of vision-based control for MAVs through a virtual flight testbed. In: International Conference on Robotics and Automation (2005)
4. Kaiser, K., Gans, N., Dixon, W.: Localization and control of an aerial vehicle through chained, vision-based pose reconstruction. In: American Control Conference, pp. 5934–5939 (2007)
5. Nelson, D.R., Barber, B., McLain, T.W., Beard, R.W.: Vector field path following for miniature air vehicles. IEEE Trans. Robot. **23**, 519–529 (2007)
6. Saunders, J., Beard, R.: Reactive vision based obstacle avoidance with camera field of view constraints. In: Guidance, Navigation, and Control Conference (2008)