Journal of Intelligent & Robotic Systems manuscript No.
(will be inserted by the editor)

Experimental Validation of a Helicopter Autopilot Design using Model-Based PID Control

Bryan Godbolt · Nikolaos I. Vitzilaios · Alan F. Lynch

the date of receipt and acceptance should be inserted later

Abstract Autonomous helicopter flight provides a challenging control problem. In order to evaluate control designs, an experimental platform must be developed in order to conduct flight tests. However, the literature describing existing platforms focuses on the hardware details, while little information is given regarding software design and control algorithm implementation. This paper presents the design, implementation, and validation of an experimental helicopter platform with a primary focus on a software framework optimized for controller development. In order to validate the operation of this platform and provide a basis for comparison with more sophisticated nonlinear designs, a PID controller with feedforward gravity compensation is derived using the generally accepted small helicopter model and tested experimentally.

1 Introduction

Unmanned Aircraft Systems research is a rapidly advancing field which has received wide interest, especially over the past two decades (see for instance [32] and the references therein). Common examples of these aircraft systems include fixed-wing, helicopters, quadrotors, and ducted fans. The latter three examples possess unique characteristics as compared to fixed wing aircraft (e.g., hover and Vertical Take-Off and Landing), however helicopters are preferred for applications which require large payload capacity or long flight endurance.

B. Godbolt · N. Vitzilaios · A. Lynch
Applied Nonlinear Controls Laboratory
Department of Electrical and Computer Engineering
University of Alberta, Edmonton, Canada, T6G 2V4
Tel.: 1-780-492-2627, Fax: 1-780-492-1811
E-mail: {godbolt, nvitzilaios, alan.lynch}@ualberta.ca

Helicopters possess underactuated, nonlinear dynamics with input coupling and therefore provide a particularly challenging control problem [26,27]. As with all applied research, experimental validation of control designs is a critical aspect of research in this field. However, developing a platform capable of conducting flight tests is a significant implementation challenge.

An experimental helicopter UAV platform consists of the airframe, avionics hardware, ground station, and the relevant software. Hardware specifications are often given in detail, and some examples include [16,19,35,38]. However, it is rare to find a similar level of detail regarding software implementation. Indeed, generally only the operating system and development tools are described. Examples of such work include platforms which run Linux [16,18–20, 22], Windows [11,35], QNX [14,21,30,37,38], VxWorks [24,29], and others [23,36]. In some cases the autopilot development tools are described such as [11,16] where MATLAB/Simulink RTW is used, and [21,22,24,30] which use ANSI C/C++.

This paper presents the experimental platform of the Applied Nonlinear Controls Lab (ANCL) at the University of Alberta, and proposes a simple control law appropriate for experimental validation. The main contribution of this paper is to introduce a novel autopilot software framework. The design of this framework provides fault tolerance, reconfigurability to allow for easy hardware changes, and a flexible environment for control algorithm implementation. Furthermore, the source for our autopilot has been released publicly and is available for download [5]. Therefore, this paper provides sufficient implementation detail to encourage other research teams to use our software. Future benefits are expected from interaction with the community in an effort to further develop this code.

In order to validate our platform, a control law was required. Since the logical progression towards fully autono-

Fig. 1 ANCL Helicopter in flight

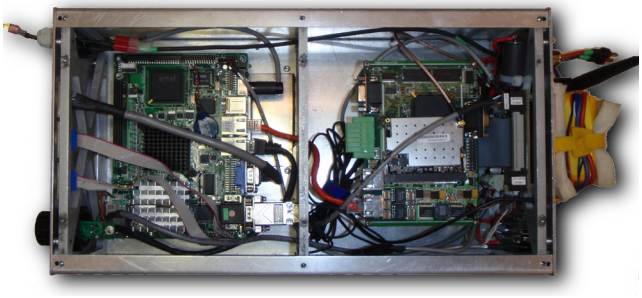

Fig. 2 ANCL Helicopter avionics

mous flight is to achieve hover stabilization, then expand the flight regime, it was only necessary to choose a control law which is valid near hover. It was therefore desirable to use a simple linear controller.

It has become common practice in the literature to design linear helicopter controllers. Recent examples include [34] and [9] which use backstepping and LQR respectively to stabilize a linear approximation of the helicopter dynamics.

Alternatively, a relatively simple approach is to use a non-model-based control law which is tuned entirely experimentally, as in [28]. The control law proposed here is similar to one which is not model-based, however it is derived using the dynamics, and known parameters are used in determining initial values for the feedback gains.

The organization of this paper is as follows. Section 2 presents the specifications for the hardware platform. In Section 3 the autopilot software is presented in detail. Section 4 presents the derivation of a proportional-integral-derivative (PID) control which is used to validate the experimental platform. Results from simulation runs and test flights are presented in Section 5. Finally, Section 6 concludes the presentation and gives future directions for platform development.

2 Hardware Platform

The ANCL Helicopter UAV, includes the Bergen Industrial Twin RC helicopter, the avionics box with an embedded computer, navigation sensors, and the Takeover Switch (TS). This platform was initially described in [25], however significant hardware changes have since been made and we will therefore give an overview of the current hardware configuration.

The Industrial Twin is powered by a two cylinder 52 cc gasoline engine with a flight endurance of 30-45 minutes depending on payload, it has a 1.8 m main rotor diameter, and is capable of lifting an approximately 10 kg payload. It has five control inputs which are actuated by five servos: main rotor lateral and longitudinal cyclic pitch, main and tail rotor collective pitch, and engine throttle. We note that these five servos correspond to actuation in only four independent degrees of freedom since the collective pitch and the throttle together influence the thrust. When the helicopter is in its stock configuration the tail rotor collective pitch is controlled by a Futaba GY-401 gyro which stabilizes the yaw angle by tracking a pilot reference. In order to circumvent the need to identify the gyro, it can be bypassed allowing direct computer control of the tail collective. For manual control a Futaba T9CHP radio is used with a Futaba R149DP receiver.

The avionics include an embedded computer, a GPS-aided inertial navigation system, a DGPS receiver, and a radio modem. The embedded computer is an Ampro ReadyBoard 800 single board computer with a Pentium M 1.4 GHz CPU, 1 GB RAM, and an 8 GB CF card for storage. The Microstrain 3DM-GX3-45 (referred to as the GX3) provides a navigation filter which is a GPS-aided INS, and an Attitude and Heading Reference System (AHRS). The navigation filter provides estimates of position, velocity, attitude and gyro bias; whereas the AHRS provides attitude estimates only. Since the integrated GPS receiver does not provide sufficient accuracy, a Novatel OEM4 FlexPak DGPS receiver (referred to as the OEM4) with 2 cm CEP position accuracy is used as an external input to the GX3. Finally, the radio modem is a 2.4 GHz Microhard VIP2400 modem (referred to as the VIP) which provides RS-232 communication for the OEM4, and Ethernet for ground station communication. The above avionics hardware is mounted in a custom made avionics box shown in Fig. 2.

The TS, which is shown in Fig. 3, allows the pilot to select the source of the servo commands as either those from the radio control receiver (Pilot Manual Mode), or the ones generated by the embedded computer (Computer Control Mode). The TS is based on the Microbotics SSA20024 Servo Switch controller, and also includes a custom designed PCB which provides connections to the receiver and the helicopter servos. Analog low pass filters have been integrated into the

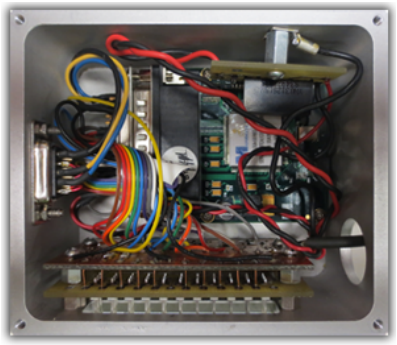

Fig. 3 Takeover Switch (TS) packaged with power supply and custom PCB

PCB in order to eliminate the noise induced by the engine in the input/output channels of the TS.

3 Autopilot Software

The purpose of the experimental platform and in particular the autopilot software is to facilitate applied control systems research for a helicopter UAV. Since this research requires frequent changes to the control algorithm as new designs are developed, a software framework is needed which allows controller implementation to be performed in an efficient manner. One possible approach would be to use Mathworks Real-Time Workshop (RTW) which generates C-code from Simulink diagrams and provides a familiar environment for developing control algorithms (see e.g., [11,16]). In fact, a prior version of our platform used a RTW package, however we found it to be insufficiently reliable, and cumbersome for hardware integration and upgrades. Furthermore, Simulink diagrams are not well-suited for collaboration (either by internal team members or the open-source community). Therefore, in order to achieve maximal flexibility as well as reliability we have chosen to develop our own software framework.

3.1 Operating System and Development Environment

The main consideration for the choice of operating system is real-time performance. While the only way to guarantee real-time performance in software may be to program a microcontroller in assembler, this approach is very labour intensive and does not scale easily to large programs. Therefore, we selected a so-called real-time operating system which provides the convenience of task scheduling and a general layer of abstraction from the hardware, but also provides features which aid the developer in meeting timing deadlines not commonly available in desktop operating systems. In particular, we chose QNX 6.5.0 [3] since it is well documented, mature and already proven in the UAV field (see for instance [38]).

Another key benefit of an operating system is access to a modern compiler suite. Since code footprint is not an issue for our hardware configuration, and the requisite compiler was provided with QNX we chose to write the autopilot in C++. Indeed, this language provides convenient high level features which promote code reuse and modularity, but can also be used for low-level programming using C syntax and libraries. In addition, there are many mature libraries for performing common yet sophisticated tasks (e.g., thread handling and linear algebra). In our implementation we make extensive use of the Boost C++ Libraries, many of which are slated to be included in upcoming C++ standards [6].

In addition to language and library dependency considerations, a development environment was devised to facilitate team collaboration. This environment includes a revisioned source repository; online HTML documentation available on the local network; and the QNX Momentics IDE which can be used to develop, compile, debug, and execute code. Indeed, an autopilot is not a trivial piece of software to design and develop, and substantial long-term benefit is envisioned by simplifying the task of contributing to the source. For example, a single centralized code repository, including revision history, allows current and future contributors to review what changes are being made to which source files and by whom. The Git source control management software [1] provides a solution for local development, and is also used to update the publicly available source. Documentation of the autopilot source is written using Doxygen markup [7]. This functionality ensures online documentation is always current and allows the documentation to be viewed without requiring a copy of the source.

3.2 Architecture

A governing principle of the software design is that it should be modular. This approach helps facilitate team collaboration and simplifies major code changes (e.g., control algorithm changes or rewriting a hardware interface when a component of the experimental platform is changed). From a high level, the software employs standard object-oriented programming techniques insofar as the code is logically divided into classes. Although there are at present many classes implemented, there are only a few main ones: a main class for high level program management and control computation timing, a class to provide a public interface for the control implementation, and a class for each piece of external hardware which contains the code used to communicate with the device.

One method employed to provide modularity in the system is use of the Boost Signals2 library. This library defines a signal type which can be dynamically connected to

(and disconnected from) one or more functions, and these functions are executed whenever the signal is "emitted." An example of how signals are used to improve modularity, is to report status changes received from the GX3. Among other things, the GX3 reports several error conditions such as large covariances for its estimated quantities. Currently, these error conditions emit signals from the class handling communication with the unit. Since it was not possible to know which functions should be called in future controllers to handle estimate errors, the only alternative to using signals which does not require future modification of the GX3 class, would be to constantly poll the GX3 class. Polling for information which is unlikely to change is inefficient (estimate errors are expected to be infrequent), however their inclusion in the public interface means that these error conditions can be handled without requiring modification of the GX3 class.

Data logging is another important aspect of the autopilot. The logging is implemented in a way which is flexible with respect to data type.[1] In addition, the logs are flushed to disk regularly to prevent memory from being entirely consumed, and to also prevent a software crash from causing a complete loss of data. The timing of these disk writes is chosen to minimize the amount of data (in memory) which would be lost in the event of a system failure, but also to ensure they are not interfering with overall system performance. Message logging is also available with varying levels of severity to provide insight into the sequence of events occurring within the autopilot. This logging facility has proven essential not only for identifying sources of failure, but also as a record of the experimental procedure during successful flights.

3.3 Controller Implementation

Since the autopilot is being developed to research advanced control designs, it must accommodate several different algorithms, possibly generating output from multiple controllers simultaneously for comparison by the operator. In order to standardize the implementation of these algorithms, all of the controller classes are derived from an abstract class named `ControllerInterface` which declares the main functions related to control computation. The control law presented in Section 4, uses an inner-outer hierarchical approach. Therefore, two controllers have been implemented: an inner loop attitude controller and an outer loop position controller. Figure 4 shows the relationship between the control classes and the `ControllerInterface` abstract class.[2] The `reset` and `runnable` functions are used to reinitialize any controller states, and test whether a controller is capable of running (e.g., whether sufficient state measurements are available), respectively. The `operator()` function performs the control computation and is distinct from the `get_control_effort` function since it may contain an integration step (as is the case for PID control). Therefore, `operator()` should only be called once per iteration of the autopilot main loop whereas `get_control_effort` can be called many times (e.g., for logging and transmitting to the ground station in addition to updating the servo commands). This approach was chosen to reflect the design philosophy of implementing each function to perform a single, clearly defined task. The result is modular code which is easier to use and maintain.

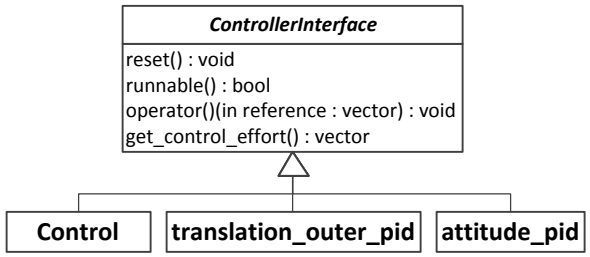

Fig. 4 UML Inheritance Diagram showing the relationship between the conrtoller classes and the abstract ControllerInterface class

The `Control` class is used to provide the rest of the autopilot with access to the controllers. It manages the various controllers by keeping track of a mode that determines which controllers to call. In addition, this class performs ancillary functions such as maintaining an XML file to store configuration parameters and generating a list of parameters which can be exchanged with the ground station. Indeed, all configuration data in the autopilot is stored using the XML format. This format is used because it provides an easy yet flexible syntax for manual editing. In addition, many libraries for interfacing with XML files exist (such as RapidXml [4] which is currently used in the autopilot).

3.4 Overview of Program Flow

An overview of the main program loop is shown in Fig. 5. This loop is executed at 100 Hz and is contained in a class named `MainApp` which is spawned as a thread upon program initiation. The decisions in this loop are strictly high level in nature, and in particular relate to selecting the source of the servo commands and waiting for threads to exit cleanly upon

[1] It is implemented using a template function which only assumes a container type with support for a Forward Iterator, and that the data in the container can be converted to `std::string` using `boost::lexical_cast`.

[2] The vector type used by `ControllerInterface` is in fact `boost::numeric::ublas::vector<double>` but is abbreviated to `vector` for presentational clarity.

program termination. This clean thread termination is performed by allowing threads to register themselves in a list. When the autopilot receives a kill command (from the operator) the `MainApp` class emits a terminate signal, the list of threads is then iterated and each element of the list is joined by the `MainApp` thread, which effectively allows the registered threads to perform any final operations before ending. In practice, examples of thread cleanup include flushing remaining data logs and closing the log files, or sending on-board hardware commands to stop transmitting data and releasing serial ports.

The control computation is performed by calling the `Control` class. The decisions made by the `Control` class are shown in Fig. 6. This operation can throw a `bad_control` exception which is handled by giving control of the servos directly to the pilot. Although this method of handling an error may not be successful in averting a disaster (e.g., the pilot may not be aware he has taken control), it should only be encountered upon a complete failure of all possible control modes. Indeed, it is preferable to use the pilot commands even without his knowledge rather than allowing an unhandled exception to cause the process to be terminated by the operating system or continuing to attempt control computation when it is no longer possible (e.g., when state measurements are no longer reliable).

The details of how the control effort is computed based on the control mode are shown in Fig. 6. Of particular interest is the built in error handling. An exception thrown by the position controller is handled by changing the mode to attitude stabilization only. The attitude control mode attempts to regulate the orientation which is tuned by the ground station operator, and should be set as closely as possible to the trim orientation. Thus, upon failure of the position controller the helicopter will attempt to stabilize to a hover. In practice the intention is that the pilot would observe an apparent failure in position control and regain manual control using the TS. However, this error handling allows the autopilot to give the pilot a chance to react by sending (hopefully) stabilizing commands to the servos.

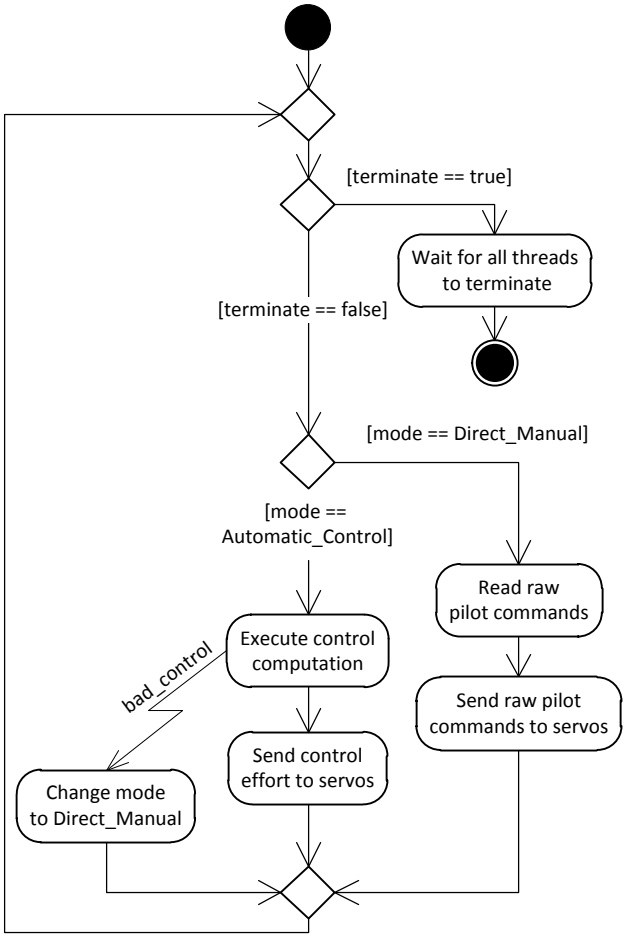

Fig. 5 UML Activity Diagram showing how the servo command source is chosen

3.5 Communication with On-board Hardware

The autopilot communicates with the GX3, OEM4, and TS (see Section 2) using RS-232. The majority of the communication with the on-board hardware involves the reception of data. Therefore in all cases a thread is spawned which reads data from the respective serial port. For the case of the OEM4, a command is only sent once when the program starts, to request that data transmission begins, and once when the program finishes to request that data transmission ceases. These commands are therefore simply implemented as part of the receiving thread. For the case of the TS, data is continually being sent as well as received. Therefore, a second thread used for sending data is executed in parallel with the receive thread. Sending commands to the GX3 is implemented slightly differently because we found that it was convenient to allow the ground station operator to have some direct control of the navigation filter. In particular, we implemented the ability for the operator to reset the navigation filter and initialize the attitude estimate based on the current AHRS measurement computed by the GX3. Since these user commands are asynchronous to the execution of the autopilot and could happen at any time (e.g., if the operator observes divergence of the state estimate he could request the pilot briefly land the helicopter while the filter is reinitialized), we designed the autopilot to spawn a thread for each command sent to the GX3. These threads do not contain a loop, instead they terminate after an acknowledgement is received from the GX3 indicating successful transmission of the command. Sequential writing to the serial port is ensured using a mutex.

Although the hardware communication essentially consists of reading and interpreting incoming serial data, along

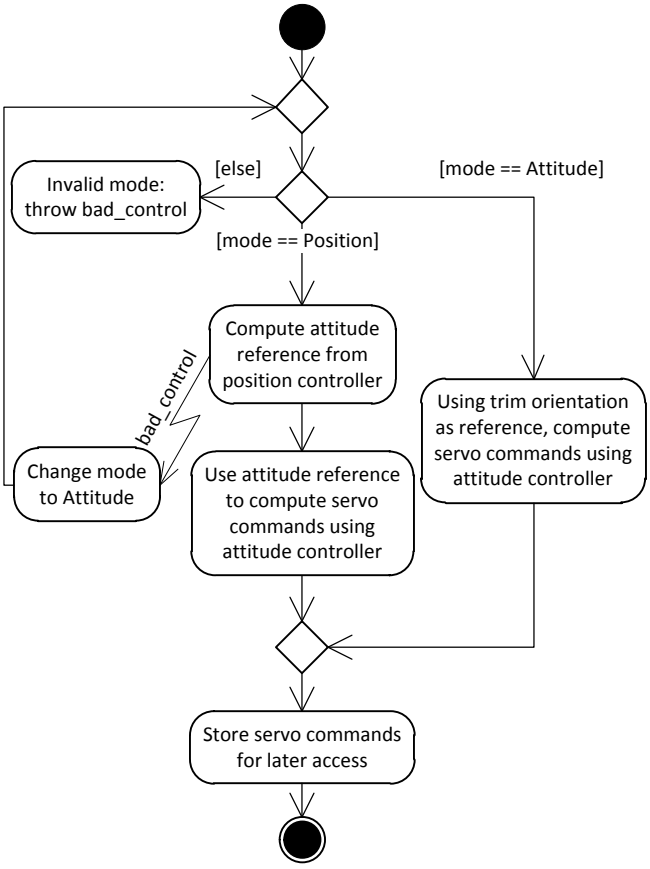

Fig. 6 UML Activity Diagram showing an overview of the control computation

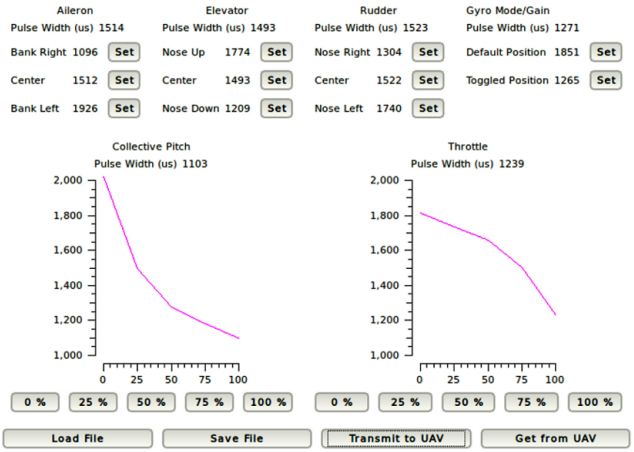

Fig. 7 Window for reading pilot radio calibration parameters which was added into QGC

with the ground station communication, it accounts for roughly 80% of the source code. This is the main reason that we chose C++ over a Simulink/RTW implementation. As is evidenced by the distribution between the code related to the control and that related to hardware communication, in terms of implementation effort it is the software framework and not the control which requires the focus. Therefore, it is most efficient to use a development environment that allows direct access to the operating system.

3.6 Ground Station

Rather than write our own ground control software we chose to modify, and contribute to, the existing open source project QGroundControl (QGC) [8]. QGC natively supports an application layer message protocol called MAVLink [2], an implementation of which is distributed alongside QGC. It is outside the scope of this document to detail QGC, however we will describe the three main modifications we made which have particular relevance to our work.

The first addition we made to QGC is a window for reading the radio calibration parameters and is shown in Fig. 7. These parameters define the transformation between the pilot stick position and the raw pulse width delivered to the servos. Due to environmental factors such as changes in temperature or wind conditions the pilot will often make adjustments to the radio calibration in the field. By using the radio calibration window, these changes can be read and updated in the autopilot without requiring recompilation of the source code, or manual configuration file editing. In practice we have found this feature invaluable during field experiments. Indeed, correct measurements of servo travel and centers are critical. This window was integrated with the official QGC source.

The second addition we made to QGC was to add a dockable window. This custom widget, shown in Fig. 8, is used to monitor and control aspects of our autopilot which are unique to our implementation. For example, this window allows the operator to view and change the servo command source, control mode, and attitude estimate source. The attitude estimate source refers to a selection between the AHRS and the navigation filter, both of which are provided by the GX3. This widget also allows the operator to view and change the location of the origin used to transform position measurements into the local navigation frame. Another function provided by this widget is to provide the interface mentioned in Section 3.5 which allows the operator to reset the navigation filter and initialize its attitude estimate from the AHRS. Finally, the operator can send a kill command to the autopilot, the result of which is described in Section 3.4.

The exchange of MAVLink messages between the autopilot and QGC is performed using UDP socket communication. This socket is accessed by a send and a receive thread which are implemented using the Boost Asio library. This link is fault tolerant in the sense that whenever possible, error conditions are detected and data retransmission is

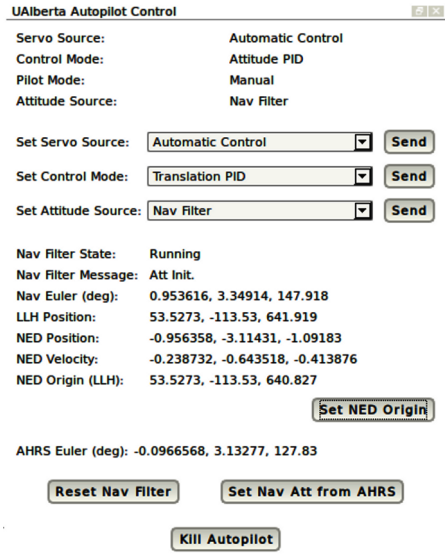

Fig. 8 QGC widget used for oversight and control of our autopilot

attempted. As much as is possible we make use of the existing MAVLink messages, however in order to fully support our extensions to QGC it was necessary to create custom messages. In QGC, the common message set is handled by the `UAS` class. The final modification we made to QGC was to specialize the `UAS` class in order to handle our custom messages.

4 PID Control Law Derivation

The purpose of the proposed control law is to provide a simple feedback capable of stabilizing the helicopter in a hover using a constant reference. We have chosen a PID control law which is similar to one which is not model-based, and thus avoids the implementation overhead of a more complex design. However, we derive our control law from the model in order to benefit from known parameters and key structural details. As compared to the PD control given in [28,39], our control law uses PID and thus includes integral feedback which can compensate for a constant disturbance. For example, in hover, gravity is a constant disturbance to the heave dynamics and PD control alone cannot be expected to converge to the reference exactly. Our control law uses model-based feedforward gravity compensation with integral feedback which will help compensate for model error.

We will proceed by approximating the helicopter dynamics and use this model to design control laws for the translational and rotational dynamics independently.

Let $\mathscr{I}$ and $\mathscr{B}$ be right-handed coordinate frames, and let $\{e_1, e_2, e_3\}$ and $\{e'_1, e'_2, e'_3\}$ be orthogonal sets of unit vectors which are aligned with the coordinate axes of $\mathscr{I}$ and $\mathscr{B}$ respectively. The navigation frame $\mathscr{I}$ is assumed inertial, has an origin fixed to the earth, and its axes point north, east, and down; $\mathscr{B}$ is fixed to the helicopter, has its origin at the helicopter's center of mass, and its axes point forward, right, and down. For a detailed treatment of the various coordinate frames commonly used for navigation see [17]. The rigid body dynamics which govern the motion of a helicopter can be expressed as (see for instance [31]),

Table 1 ANCL Helicopter parameters

m	13.7 kg
J_x	0.36 kg m^2
J_y	1.5 kg m^2
J_z	1.2 kg m^2
l_M	$(0, 0, -0.32)$ m
l_T	$(-1.06, 0, -0.12)$ m

$$\dot{p} = -\omega \times p + v \tag{1a}$$
$$m\dot{v} = -\omega \times mv + f \tag{1b}$$
$$\dot{\eta} = W\omega \tag{1c}$$
$$J\dot{\omega} = -\omega \times J\omega + \tau \tag{1d}$$

where p is position, v is velocity, m is mass, f is applied force, $\eta = (\phi, \theta, \psi)^T$ is the orientation expressed using roll-pitch-yaw Euler angles, W is the transformation between the body-fixed angular velocities and the derivatives of the Euler angles, ω is the body-fixed angular velocity, J is the inertia matrix, and τ is the applied torque. All state variables (with the exception of η) are expressed in the body frame $\mathscr{B}$. A generally accepted model of applied force and torque is established in the literature (e.g., [15,27]):

$$f = \begin{pmatrix} -T_M a \\ T_M b - T_T \\ -T_M \end{pmatrix} + mgR^T e_3 \tag{2a}$$
$$\tau = l_T \times \begin{pmatrix} 0 \\ -T_T \\ 0 \end{pmatrix} + l_M \times \begin{pmatrix} -T_M a \\ T_M b \\ -T_M \end{pmatrix} + \begin{pmatrix} 0 \\ -Q_T \\ -Q_M \end{pmatrix} \tag{2b}$$

where T_M, and T_T are the main and tail rotor thrusts respectively; a and b are the longitudinal and lateral main rotor flapping angles respectively; g is the acceleration due to gravity; $R \in SO(3)$, $R : \mathscr{B} \to \mathscr{I}$ is the rotation matrix transforming vectors expressed in the body frame into vectors expressed in the inertial frame and is parametrized by η; l_M and l_T are the positions of the main and tail rotor hubs respectively, each expressed in $\mathscr{B}$; Q_M and Q_T are the main and tail countertorques respectively, they are non-negative and oriented in the direction of blade rotation for our helicopter. Table 1 shows the values of the model parameters for the ANCL Helicopter, where the mass of the helicopter is measured without fuel, and $J = \mathrm{diag}(J_x, J_y, J_z)$.

The contributions of $T_M a$, $T_M b$, and T_T to the translational force are often called the small-body forces (SBF). It is common practice to ignore the effect of the SBF for the purpose of control design [27,15,33]. In addition to removing the SBF we make the approximations: $l_M \approx (0,\, 0,\, -z_M)^T$, and $l_T \approx (-x_T,\, 0,\, 0)^T$ where z_M and x_T are the distances between the center of mass and the main and tail rotors in the e_3' and e_1' directions respectively. In general the countertorques Q_M and Q_T are functions of thrust and near hover they will be approximately constant. Therefore, we consider Q_M and Q_T to be unknown constants which will be ignored in the derivation of the control and later compensated by integration.

We observe that the equilibrium of (1) where the SBF have been removed from (2) is $\omega = v = 0$, $\theta = \phi = 0$, where p and ψ are unconstrained. Using this equilibrium, we will take a first order linear approximation of the above dynamics with the input transformations:

$$u^t = (-g\theta,\, g\phi,\, \frac{-T_M}{m} + g)^T \tag{3a}$$

$$u^r = J^{-1}(z_M T_M b,\, z_M T_M a,\, -x_T T_T)^T \tag{3b}$$

which represent the translational and rotational inputs respectively. Applying these operations and combining (1) and (2), we obtain

$$\dot{p} = v \tag{4a}$$

$$\dot{v} = u^t \tag{4b}$$

$$\dot{\eta} = \omega \tag{4c}$$

$$\dot{\omega} = u^r \tag{4d}$$

We define position and heading reference trajectories p^* and ψ^*. We now partition the dynamics into translational and rotational subsystems. Beginning with the translational dynamics we define error states $\tilde{p} = p - p^*$, $\tilde{v} = v - \dot{p^*}$. We write the error dynamics as

$$\dot{\xi}^t = \tilde{p} \tag{5a}$$

$$\dot{\tilde{p}} = \tilde{v} \tag{5b}$$

$$\dot{\tilde{v}} = \tilde{u}^t \tag{5c}$$

where $\tilde{u}^t = u^t - \ddot{p^*}$ and we have added an integrator state ξ^t. These error dynamics can be stabilized by

$$\tilde{u}^t = -k_d^t \tilde{v} - k_p^t \tilde{p} - k_i^t \xi^t \tag{6}$$

which defines an input that can be transformed into the main rotor thrust and a reference roll-pitch to be regulated by the attitude control.

Recalling (3a) we note that an extra term remains in the relationship between thrust and the input defined by (6) due to gravity. Since the gravity term is constant it could be ignored as was done with the countertorques and compensated by integration. However, unlike the countertorques this term is known precisely, thus rather than wait for an integrator state to build up error we chose to compensate it directly. Inverting the input transformation for the main rotor thrust and assuming a constant reference gives $T_M = -m\tilde{u}_3^t + mg$. To proceed we must relate the main rotor thrust to the (normalized) main rotor collective pitch which is the physical input to the system. Some examples of thrust to collective pitch relationships can be found in [12,13,28]. However, in general the induced velocity of the main rotor causes the thrust to collective equation to not be solvable algebraically. Therefore, we chose a simplified relationship which uses a hover assumption to remove the translational and rotational velocities [10] which reduces the expression for main rotor thrust in terms of collective pitch to

$$T_M = CA_M + \frac{D^2}{2} - D\sqrt{CA_M + \frac{D^2}{4}} \tag{7a}$$

$$C = \frac{\rho a c R^3 N_b \Omega^2}{6} \tag{7b}$$

$$D = \frac{acRN_b\Omega\sqrt{\rho}}{4\sqrt{2\pi}} \tag{7c}$$

where A_M is the normalized main rotor collective pitch, ρ is the air density, a is lift curve slope, c is chord length, R is rotor disk radius, N_b is the number of blades and Ω is the rotor speed. The parameters in (7) are given in [10] for the ANCL helicopter. This reference also contains full details on the helicopter model.

All that remains is to define the attitude control. We thus consider η^* to be the reference generated by the translational control augmented by ψ^*. We define $\tilde{\eta} = \eta - \eta^*$, and $\tilde{\omega} = \omega - \dot{\eta}^*$ which results in the attitude error dynamics

$$\dot{\xi}^r = \tilde{\eta} \tag{8a}$$

$$\dot{\tilde{\eta}} = \tilde{\omega} \tag{8b}$$

$$\dot{\tilde{\omega}} = \tilde{u}^r \tag{8c}$$

where $\tilde{u}^r = u^r - \ddot{\eta}^*$ and we choose the stabilizing control law $\tilde{u}^r = -k_d^r \tilde{\omega} - k_p^r \tilde{\eta} - k_i^r \xi^r$. Since the countertorques are assumed constant in (2) the integral term will compensate for them in steady state.

5 Experimental Procedure and Results

Before testing the control law experimentally, the system response was simulated in order to verify the design. The form of the actuation for the simulated plant was kept general as given in (2). In other words the small-body forces and the countertorques were included, and the rotor hub positions were kept general. The values used for the countertorques were $Q_i = 0.02|T_i|,\ i \in \{M, T\}$. The rotor hub offsets and the inertial parameters were as given in Table 1. The simulated state response is shown in Fig. 9, and the control effort

Table 2 Controller gains used for the simulation

k_d^t	diag(1,1,10)
k_p^t	diag(1,10,3)
k_i^t	diag(3,3,1)
k_d^r	diag(5,3,3)
k_p^r	diag(17,20,20)
k_i^r	diag(10,5,5)

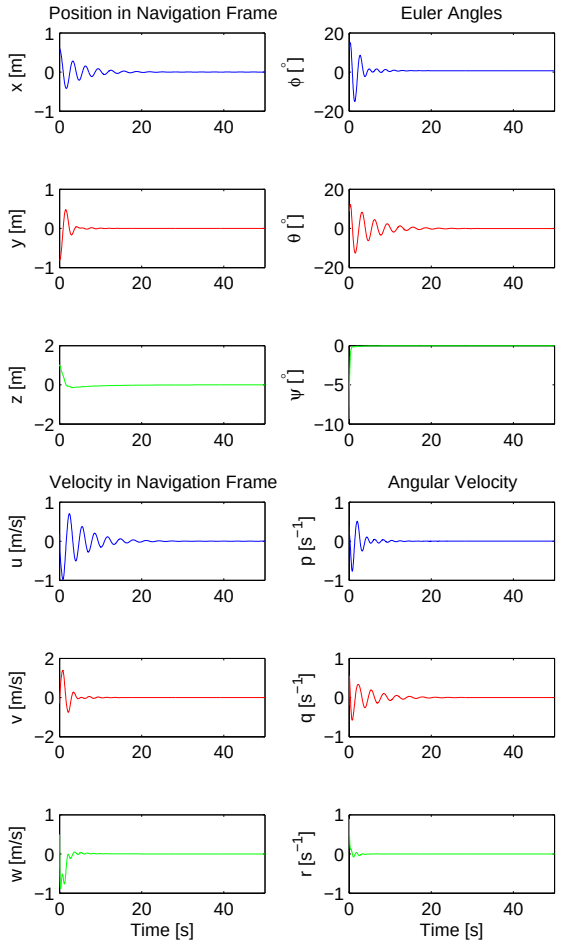

Fig. 9 State response when simulated plant includes small body forces and countertorques

is shown in Fig. 10. In order to obtain the desired transient, we tuned the gains to provide faster response in the rotational dynamics than the translational dynamics. The controller gains which were used are given in Table 2.

We chose to perform the experimental validation incrementally by first testing the control in the roll-pitch directions using the rotational controller. For the initial flights

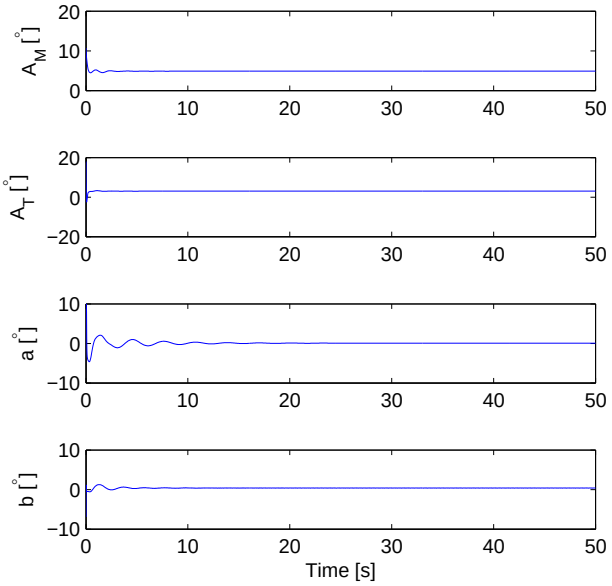

Fig. 10 Control effort for the simulation

we wanted controller gains which would not cause the helicopter to react aggressively. Thus, we used standard linear systems time domain specifications to obtain initial values for the control gains. However, in order to apply the control signal to the helicopter we required a relationship between the flapping angles which are taken as the inputs in (2b), and the (normalized) cyclic inputs which can be converted into servo commands using the radio calibration (see Section 3.6). In general, the dominant means of actuating the flapping angles is a mixture of the direct feedthrough from the cyclic inputs, with the flapping dynamics of the flybar. We approximate this relationship by taking the steady state response of the flybar dynamics assuming no angular velocity

$$a = k_B \delta_p + k_F k_H \delta_p$$
$$b = k_B \delta_r + k_F k_H \delta_r$$

where δ_r and δ_p are the roll and pitch cyclic inputs respectively, and $k_B = 0.09$ rad, $k_H = 0.66$, $k_F = 0.28$ rad, are the mechanical gains associated with the Bell-Hiller linkages.

Thus, assuming a constant reference we obtain a transformation between the controller gains given in the input space used for the control design (as defined in (3)) and the physical inputs

$$\begin{pmatrix} \delta_r \\ \delta_p \end{pmatrix} = \frac{1}{(k_B + k_F k_H)} \begin{pmatrix} \frac{J_x}{z_M m g} & 0 \\ 0 & \frac{J_y}{z_M m g} \end{pmatrix} (-\bar{k}_d^r \bar{\omega} - \bar{k}_p^r \bar{\eta} - \bar{k}_i^r \bar{\xi}^r) \tag{9}$$

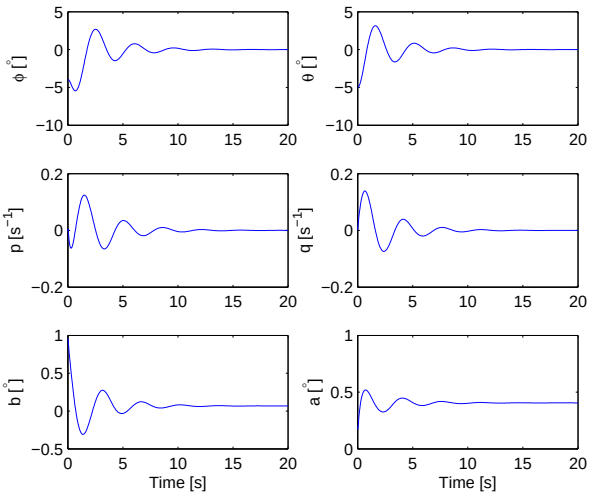

Fig. 11 Simulated attitude response using initial controller gains

where the main rotor thrust is assumed to be equal to the force of gravity, and the overbar is used to emphasize the fact that only the roll-pitch error states and controller gains are considered. In order to obtain a conservative response we chose a rise time of 1 s and a damping ratio of 0.3. The third pole of the closed loop dynamics was taken to have a large enough real part to minimize its effect on the transient response. The initial values for the controller gains were $\tilde{k}_p = \text{diag}(0.26, 1.1)$, $\tilde{k}_d = \text{diag}(0.18, 0.75)$, $\tilde{k}_i = \text{diag}(0.48, 2.0)$, where $\tilde{\cdot}$ denotes the roll-pitch controller gains given in the physical input coordinates. The simulated attitude response using these gains is shown in Fig. 11.

Since the simulation results showed a response which appeared safe to test on the helicopter (i.e., not overly aggressive), we proceeded with the experimental validation. The reference was a constant trim which was tuned along with the controller gains using QGC. In the ideal case, this trim is the equilibrium of (1), and in particular a small positive roll angle is necessary to offset the translation force created by the tail rotor. The servo connections were configured such that only the cyclic servos passed through the TS. Thus, the pilot retained complete control of the main and tail rotor thrusts.

For initial tests, as a safety measure the attitude controller was not given full authority of the servo commands. Instead, a weighted mean was taken of the controller output and the pilot commands. This weighting was then changed on the roll and pitch channels individually (using QGC) to reduce the pilot's influence until ultimately the autopilot was given full control.

Results showing the system response to the autopilot are given in Fig. 12, and the gains which were used are given in Table 3. The plots show the difference between the orientation and the trim, the angular velocity, and the control effort for the roll and pitch channels respectively. The autopilot is engaged between the vertical lines. The cyclic channels show the pilot's control signal while the pilot is in control, and the autopilot control signal when the autopilot is in control. During the first 10 s, the pilot is hovering the helicopter in order to prepare to switch into Computer Control Mode. This hover is perceived by observers on the ground to be stable. However, it is clear from the state measurement that under manual control the helicopter is constantly oscillating with an amplitude of approximately 3°-5° as the pilot corrects for drift, whereas the autopilot holds the orientation closer to the trim.

Table 3 Controller gains used for experimental testing

k_p	$\text{diag}(1,1)$
k_d	$\text{diag}(0.5,0.8)$
k_i	$\text{diag}(0.8,0.5)$

While the system is in Computer Control Mode, although it is apparent that the autopilot is driving the attitude error to zero, it is slow in doing so. This response time is was designed to moderate the state trajectories during testing, however it can be reduced by increasing the controller gains. The pitch error shows the system stabilizing to approximately 2° of error. However, we see the control signal increasing as the error is integrated. This interaction between the steady state error and the integrator state indicates correct operation of the attitude control.

Since only the orientation was being controlled, the helicopter began to translate, and after 11.2 s of autopilot control the helicopter had drifted 10 m. Thus, the pilot took back control for safety. As a result, there is an abrupt change in attitude during the last 1.5 s due to the pilot flying the helicopter to a different position.

6 Conclusions

This paper described the successful experimental validation of the autopilot software design used to control the ANCL Helicopter UAV. Fault tolerance is provided by the error handling built into the control computation as well as by the implementation of the link to the ground station. Reconfigurability is achieved by the modularity of the software design and hierarchical class structure. The flexibility of the software is ideal for rapid control algorithm development. Experimental results showing the system response to the attitude controller are provided.

Future work includes validating the translational controller which has already been implemented. In addition, we intend to perform experiments where a time-varying reference is used in order to investigate the effects of time-

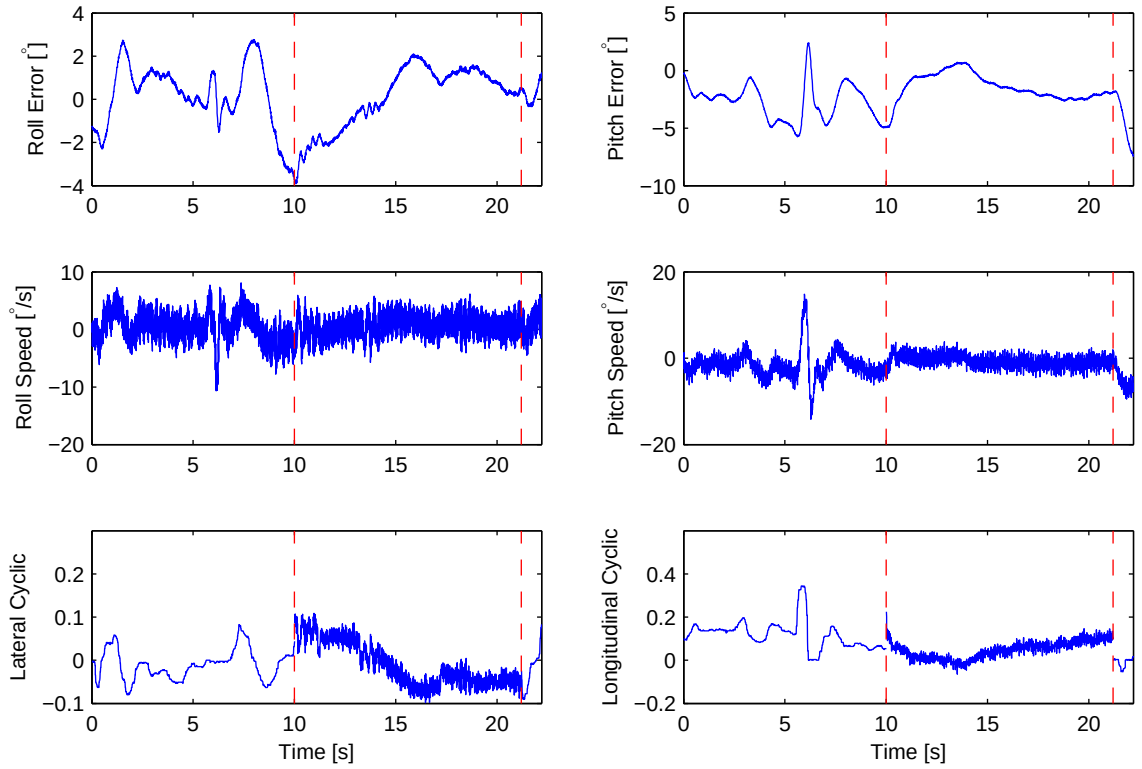

Fig. 12 Experimental results showing attitude controller response for a reference trim of $\phi^* = 0.8°$ and $\theta^* = 0°$, where the cyclic inputs are normalized and the autopilot is engaged during the time between the vertical (red) lines

varying countertorques and the SBF. Furthermore, novel nonlinear control designs will be tested using the platform.

Acknowledgements

The authors would like to thank Andre Nadon, RC helicopter pilot, for maintaining and piloting the helicopter. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Strategic Grants Program.

References

1. Git - Fast Version Control System. http://git-scm.com/. Accessed 18 June 2012
2. Micro Air Vehicle Communication Protocol. http://qgroundcontrol.org/mavlink/start. Accessed 18 June 2012
3. QNX Software Systems Online Infocenter. http://www.qnx.com/developers/docs/6.5.0/index.jsp. Accessed 18 June 2012
4. RAPIDXML Manual. http://rapidxml.sourceforge.net/manual.html (2009). Accessed 18 June 2012
5. ANCL/autopilot - Github. http://github.com/ANCL/autopilot (2012). Accessed 18 June 2012
6. Boost 1.46.1 Library Documentation. http://www.boost.org/doc/libs/1_46_1/ (2012). Accessed 18 June 2012
7. Doxygen. http://doxygen.org (2012). Accessed 18 June 2012
8. QGroundControl. http://qgroundcontrol.org/ (2012). Accessed 18 June 2012
9. Ahmed, B.: Autonomous Landing of Lightweight Helicopters on Moving Platforms such as Ships. Ph.D. thesis, The University of New South Wales at The Australian Defence Force Academy (2009)
10. Barczyk, M.: Nonlinear state estimation and modeling of a helicopter uav. Ph.D. thesis, Dept. of Electrical and Computer Engineering, University of Alberta, Edmonton, AB (2012)
11. Bernard, M., Kondak, K., Hommel, G.: Framework for Development and Test of Embedded Flight Control Software for Autonomous Small Size Helicopters. In: Embedded Systems - Modeling, Technology, and Applications, pp. 159–168. Springer-Verlag (2006)
12. Bilal Ahmed, H.R.P., Garratt, M.: Flight control of a rotary wing UAV using backstepping. Int. J. Robust Nonlinear Control **20**, 639–658 (2010)
13. Bisgaard, M.: Modeling, Estimation, and Control of Helicopter Slung Load System. Ph.D. thesis, Aalborg University (2007)
14. Cai, G., Peng, K., Chen, B.M., Lee, T.H.: Design and Assembling of a UAV Helicopter System. In: International Conference on Control and Automation (ICCA2005), pp. 697–702. Budapest, Hungary (2005)

15. Castillo, P., Lozano, R., Dzul, A.E.: Modelling and Control of Mini-Flying Machines. Springer-Verlag, London (2005)
16. del Cerro, J., Barrientos, A., Artieda, J., Lillo, E., Gutierez, P., Martin, R.S.: Embedded Control System Architecture applied to an Unmanned Aerial Vehicle. In: IEEE International Conference on Mechatronics, pp. 254–259. Budapest (2006)
17. Farrell, J.A.: Aided Navigation: GPS with High Rate Sensors. McGraw-Hill (2008)
18. Ferruz, J., Vega, V., Ollero, A., Blanco, V.: Embedded control and development system for the HERO autonomous helicopter. In: IEEE International Conference on Mechatronics, pp. 1–6. Malaga, Spain (2009)
19. Garcia, R., Valavanis, K.: The Implementation of an Autonomous Helicopter Testbed. J. Intell. Robot. Syst. **54**(1–3), 423–454 (2008)
20. Garratt, M., Ahmed, B., Pota, H.R.: Platform Enhancements and System Identification for Control of an Unmanned Helicopter. In: 9th International Conference on Control, Automation, Robotics and Vision, pp. 1981–1986. Singapore (2006)
21. Gavrilets, V., Frazzoli, E., Mettler, B., Piedmonte, M., Feron, E.: Aggressive Maneuvering of Small Autonomous Helicopters: A Human-Centered Approach. Int. J. Robot. Res. **20**(10), 795–807 (2001)
22. Geng, W., Huanye, S., Tiansheng, L.: Development of an Embedded Intelligent Flight Control System for the Autonomously Flying Unmanned Helicopter *Sky-Explorer*. In: Embedded Systems - Modeling, Technology, and Applications, pp. 121–130. Springer-Verlag (2006)
23. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W.: From Control Models to Real-Time Code Using Giotto. IEEE Control Syst. Mag. **23**(1), 50–64 (2003)
24. Johnson, E.N., Schrage, D.P.: The Georgia Tech Unmanned Aerial Research Vehicle: GTMax. In: AIAA Guidance, Navigation, and Control Conference and Exhibit. Austin, TX, USA (2003)
25. Kastelan, D.R.: Design and Implementation of a GPS-aided Inertial Navigation System for a Helicopter UAV. Master's thesis, Dept. of Electrical and Computer Engineering, University of Alberta, Edmonton, AB (2009)
26. Kendoul, F.: Survey of Advances in Guidance, Navigation, and Control of Unmanned Rotorcraft Systems. J. Field Robot. **29**, 315–378 (2012)
27. Koo, T.J., Sastry, S.: Output Tracking Control Design of a Helicopter Model Based on Approximate Linearization. In: 37th IEEE Conference on Decision and Control, pp. 3635–3640. Tampa, FL (1998)
28. Mettler, B.: Identification Modeling and Characteristics of Miniature Rotorcraft. Kluwer Academic Publishers, Norwell, MA (2003)
29. Mettler, B., Tischler, M.B., Kanade, T.: System Identification of Small-Size Unmanned Helicopter Dynamics. In: Annual Forum Proceedings-American Helicopter Society, pp. 1706–1717. Montreal, Canada (1999)
30. Montgomery, J.F., Johnson, A.E., Roumeliotis, S.I., Matthies, L.H.: The Jet Propulsion Autonomous Helicopter Testbed: A Platform for Planetary Exploration Technology Research and Development. J. Field Robot. **23**(3-4), 245–267 (2006)
31. Murray, R.M., Li, Z., Sastry, S.S.: A Mathematical Introduction to Robotic Manipulation. CRC Press, Boca Raton, FL (1994)
32. Ollero, A., Merino, L.: Control and perception techniques for aerial robotics. Annu. Rev. Control **28**(2), 167–178 (2004)
33. Raptis, I.A., Valavanis, K.P., Moreno, W.A.: A Novel Nonlinear Backstepping Controller Design for Helicopters Using the Rotation Matrix. IEEE Trans. Control Syst. Technol. **19**, 465–473 (2011)
34. Raptis, I.A., Valavanis, K.P., Vachtsevanos, G.J.: Linear tracking control for small-scale unmanned helicopters. IEEE Trans. Control Syst. Technol. **20**, 995–1010 (2012)
35. Remuss, V., Musial, M., Deeg, C., Hommel, G.: Embedded System Architecture of the Second Generation Autonomous Unmanned Aerial Vehicle MARVIN MARK II. In: Embedded Systems - Modeling, Technology, and Applications, pp. 101–110. Springer-Verlag (2006)
36. Roberts, J.M., Corke, P.I., Buskey, G.: Low-Cost Flight Control System for a Small Autonomous Helicopter. In: IEEE International Conference on Robotics and Automation, pp. 546–551. Taipei, Taiwan (2003)
37. Saripalli, S., Montgomery, J.F., Sukhatme, G.S.: Visually Guided Landing of an Unmanned Aerial Vehicle. IEEE Trans. Robot. Autom. **19**(3), 371–380 (2003)
38. Shim, D.H., Kim, H.J., Sastry, S.: Hierarchical Control System Synthesis for Rotorcraft-Based Unmanned Aerial Vehicles. In: AIAA Guidance, Navigation, and Control Conference and Exhibit. Denver, CO, USA (2000)
39. Shim, H.: Hierarchical Flight Control System Synthesis for Rotorcraft-based Unmanned Aerial Vehicles. Ph.D. thesis, Dept. of Mechanical Engineering, University Of California, Berkeley, CA (2000)