Communication, Navigation and Control using FlightGear Simulations

Aerospace 450: Flight Software Systems
December 20th, 2012

Prepared by:
Duncan Miller
Hrishi Shelar
Joshua Thomas
Aerospace Engineering
University of Michigan

Abstract. With the growing demand for Unmanned Arial Vehicles in both military and commercial applications comes a proportional increase in demand to test the flight algorithms in a safe, repeatable environment. The purpose of this project is to facilitate the development of these algorithms by providing a C++ software testbed to optimize proposed controllers. We have successfully implemented a purely custom flight control system that communicates with a 3D simulator (FlightGear) and can stabilize a standard fixed wing aircraft to steady level flight from a reasonably disturbed position. Moreover, we have proven basic maneuvering through steady level turns and climbs. Now, guidance, navigation and controls engineers are able to quickly and easily test fixed wing flight path algorithms in a purely open source environment – this capability has never been implemented at the University of Michigan to our knowledge.

Table of Contents

1 Introduction and Motivation for Research 3
2 Project Summary 3
3 Augmentation and Evaluation of UAV Playground 4
4 Transition to C++ for Stabilization 6
5 Multi-Threaded Architecture 7
6 Socket Communication Between Processes 8
6.1 Communication Protocol 8
7 Data Logging and Data Sharing 9
8 GPS Parsing and Recording 10
9 Terminal Input 10
10 Control Architecture 11
10.1 Stabilization 11
10.2 Steady Flight Maneuvers 14
11 System Performance 15
12 Education Value 15
13 Broader Impacts 16
Appendix A: Original Proposal Schematic 17
Appendix B: Procedure for Running our Final Project 18

1 Introduction and Motivation for Research

As autonomous, unmanned aerial vehicles (UAVs) begin to operate regularly in the National Airspace System, the ability to safely test the coordination and control of flight vehicles will be an important capability. Taking the pilot 'out of the loop' has both civilian and military applications. For example, the U.S. government often needs to remotely track high profile military targets—UAVs can be deployed to follow and observe transactions with little risk to human life. Similarly, UAVs can be used to survey suspicious areas of land or actively guard valuable assets during transfer. Freight transportation can also be optimized to avoid adverse weather conditions in real time. These are the applications that our project will enable.

This team has worked to establish an autonomous vehicle testbed that will allow sophisticated testing of control algorithms in a protected, repeatable environment. Evaluating the reactions of air vehicles in a simulated environment reduces time and cost, while allowing the user to log, replay and explore critical events with greater precision.

2 Project Summary

Our submitted flight controller is able to stabilize a modeled aircraft and perform basic maneuvering given simulated data from FlightGear. Originally, we proposed a three layer abstraction process to design a robust flight path controller. The process of control for any aircraft starts with the physical movement of the control surfaces. The position of these control surfaces is handled by a PID controller. Once tuned, maneuvers such as banking and ascending can be established. These maneuvers are the building blocks to complex flight paths. Hence there are three layers of control, namely:

1. Inner Loop Stabilization
2. Basic Maneuvering
3. Advanced Flight Path

Available open-source software aided with the first stage, but not to the extent originally intended. FlightGear is the free, open source flight simulator chosen for this project. It has the capabilities to model a wide variety of aircraft and ground vehicles quite realistically. FlightGear bridges real world tests with more complex possible scenarios, emulating sensors and state data such as GPS for our autopilot to parse and utilize. FlightGear provides a virtual environment with real feedback in which we can 'fly' our UAV. Open-source controllers also exist. UAV Playground is an open source Java application that we intended to adapt for optimal outer loop control of UAVs. It is a front-end application that connects to FlightGear using a socket connection. Due to a supposed memory leak with UAV Playground, however, we ultimately built a flight controller exclusively in C++ that succeeded in emulating everything in the released version of UAV Playground with additional maneuvering capabilities.

Our C++ controller is able to feed realtime data generated by FlightGear through virtual PID control loops to stabilize the flight. The controller is able to interpret real-time user input and perform the desired maneuvers. Our controller also generates time-stamped NMEA files that can be parsed into Google Earth kml files and post processed after flight termination to verify maneuvering accuracy.

3 Augmentation and Evaluation of UAV Playground

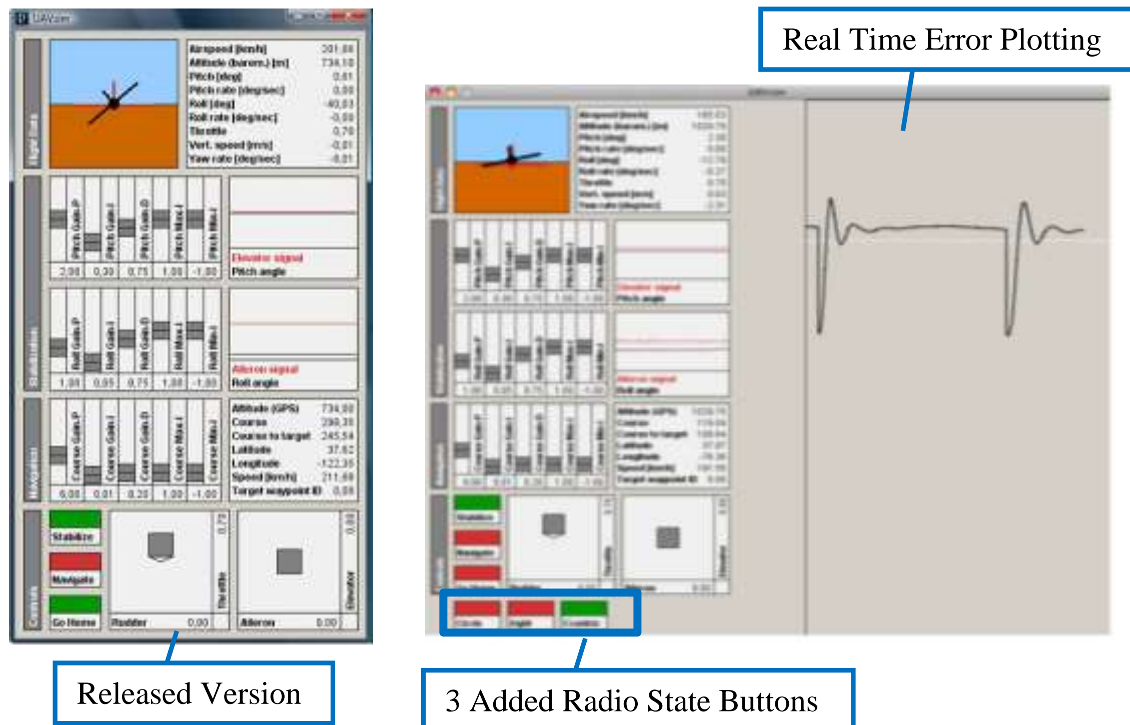
FlightGear is an open source, multi-platform flight simulator that has been in development since the 1990s. Researchers in academia, industry experts, and even hobbyists have used it to model aircraft, assess flight path performance, and on occasion have some fun. FlightGear is an attractive system for many reasons. First, the simulation engine, SimGear, allows for the user to customize a wide range of aircraft—gliders, rotary wing aircraft, personal planes, and airliners. This also enables proposed aircraft designs to be crudely tested in a simulated environment prior to production. Second, the flight dynamics model (FDM) is a high fidelity, non-linear, six degree-of-freedom simulation application that combines three parallel, independent data models—JSBSim, YASim and a derivative of UIUC. The FDM analyzes forces and disturbance torques to calculate aircraft motion with outstanding accuracy. In fact, the atmosphere is realistic enough to update with the current local weather and includes details such as eddy wind currents which can be toggled.

Screen captures of the FlightGear virtual interface

As a state simulator, FlightGear inherently lacks a flight controller. For hobbyists (or even pilots in mock cockpits), control input comes from the user through joystick voltage measurements. However, real-time attitude control is best automated through a programmable software controller that can respond with a qualitatively "high" frequency. Thus, FlightGear was designed from the beginning for dominantly open socket communication. Custom control interfaces can then be configured to receive state data and transmit control outputs. Many such FlightGear interfaces have been constructed and released both commercially and as open-source projects. One example is UAV Playground. However, few (if any) have been implemented at the University of Michigan Ann Arbor campus.

UAV Playground is an open source Java applet that was Beta released in 2009. It allows for the real time control of simulated aircraft through stabilizing PID control loops. The socket protocols between FlightGear and UAV Playground are already defined, which is why we originally chose to work directly from this platform. Using UAV Playground allowed us to "black-box" both the controller-simulator interface and the inner loop stabilization controller. This would enable our team to begin testing flight algorithms immediately.

We made great progress in developing flight maneuvers. Step two in our proposed project, after inner loop stabilization and prior to flight path algorithms, was to program basic maneuvers in flight—steady level turns, steady climb, steady descent. After reviewing the provided source code, we were able to design and successfully implement two significant features to the UAV Playground application: flight maneuvers (single circle, concentric circling, figure eight) and real time error plotting.



Original UAV Playground GUI compared with our augmentations

Two principle java files were modified for our augmentations to UAV Playground: *UAVsim.java* and *MissionController.java*. By expanding the applet window, we were able to add radio buttons that toggled on three new flight states: Circle, Eight, and Concentric. Circling is coordinated by recording the current flight heading and banking to a specified roll of 20 degrees until matching the initial heading to within 5 degrees, at which point it levels. The figure eight is almost equivalent, except that the instead of leveling, it banks a negative 20 degrees until re-matching the initial heading. Finally, the concentric circle records the initial heading and the initial position. After the first circle, the plane flies level until reaching the new radius (+200 meters) away from the initial position. A PID loop over the roll angle ensures that plane follows each new specified radius (sample error is plotted in the figure above). These are implemented in the two submitted code files and our augmentations are marked by "//Aero 450 Edits Begin" and "//Aero 450 Edits End". For the sake of simplicity, the rest of the UAV Playground source code was not submitted, but can be found online at: http://code.google.com/p/uavplayground/. Thus, our algorithms for basic maneuvering were proven to work. The figure below shows a data sample of our concentric circling flight path.

Sample concentric flight path from UAV Playground plotted in Google Earth

Many valuable software skills were learned throughout this entire process. First, we studied the accepted communication protocols from FlightGear and how to properly read and transmit state data. Second, we learned how to read and write object-oriented Java code—our course lectures in OOP helped with this considerably. In order to recompile the original UAV Playground source code, our team had to learn the Eclipse IDE for Java developers. This included creating new projects, properly referencing source/libraries, and debugging. Finally, we gained experience with the Processing language and IDE. Processing can compile java applets and uses its own language, which combines elements of Java with Arduino. We attempted to use Processing as an alternative to Eclipse following the suspected memory leak issues discussed next.

4 Transition to C++ for Stabilization

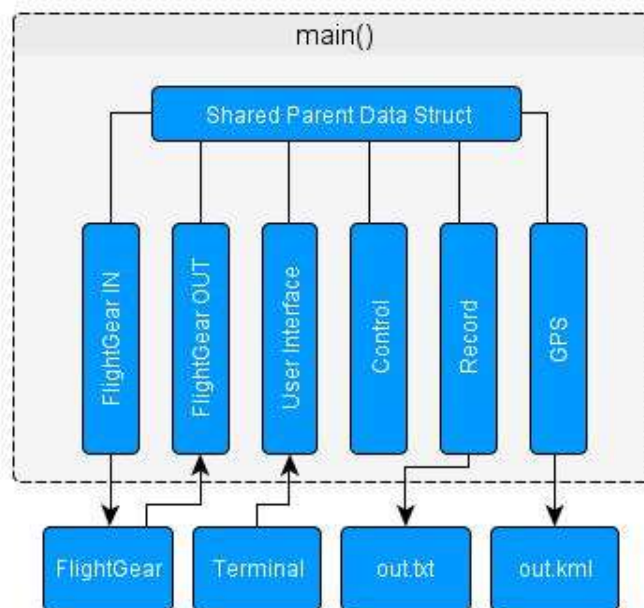
Within the scope of UAV Playground, our team was unable to move past basic maneuvering to flight path due to a presumed memory leak in UAV Playground. While working on the dual boot A2SYS computer, UAV Playground was compiled and run on the Linux platform but unexpectedly crashed after approximately 30 seconds of run time. We observed the same behavior when we restarted our project in Windows. Significant effort was put forth to identify and mitigate the potential memory leak. We observed the computer's dynamic memory overflowing before FlightGear crashed (requiring an OS restart). Although we did not have the capacity to run FlightGear and UAV Playground on our laptops, we were able to successfully compile and run on a Mac OS X Snow Leopard over Thanksgiving break—the UAV Playground data presented here was collected on a Mac.

However, with only 30 seconds of run time available on campus (the Windows/Linux A2SYS machine), we decided to instead transition to a C++ implementation of UAV Playground.
Creating a custom flight controller in C++ had many advantages. First, we were able to apply all of the C++ methods presented in class (classes, sockets, threads). Also, it resulted in a much more satisfying final product, since all of the code (with the exception of simple_sock) was written from scratch—nothing in our project is "black-boxed." We were able to use much of the experience gained from our UAV Playground adventures (specifically XML protocol) in designing our C++ architecture.

This resulted in a slight project de-scope approved by Professor Atkins. The communication and stabilization functionality, originally black-boxed and considered complete, needed to be re-created from scratch in C++. This cost us valuable development time and advanced flight path algorithms were no longer considered. Our C++ controller was only required to stabilize and maneuver (steady level, turning and climbing flight)—this was demonstrated through video during the presentation.

5 Multi-Threaded Architecture

Our C++ FlightGear controller was designed with simplicity in mind. We run a single main() function with six parallel threads, each with a different purpose and operating period as shown in the figure below. Our FlightGearIN and FlightGearOUT threads transmit and receive state data respectively at 10 Hz. The User Interface thread waits for new character commands from the user. The Control thread reads the shared data using mutexes, runs our three PID loops (the three rotational axes) and writes the commanded control outputs to the same shared data struct. Our Record thread output state data (tab delimited) to a new text file for the duration of the test. Finally, our GPS logging thread records the NMEA sentences as they are output by FlightGear.



The 'parent' data struct is shared between all threads

The shared data is a single struct passed as a void pointer to all simultaneous threads. This parent struct included two double buffers (fg_in and fg_out) and the User Interface struct (UI), which includes the mode (a character), an associated value for that mode, and a record toggle. Since it is a pointer, all the threads access the same block of memory and threads read the most accurate state numbers. Mutexes prevent errors in reading and writing to this memory block.

Header (.h) Main (.cpp)

```
struct UIs {
      char mode;
      double value;
      bool record;
};

struct parent {
      double fg_in[13];
      double fg_out[4];
      UIs UI;
};
```

```
int main() {
      pthread_t receive;
      pthread_t input;
      pthread_t control;
      pthread_t transmit;
      pthread_t record;

      parent data;

      iret1 = pthread_create(&receive, NULL,
      &receivef, (void *) &data);

      iret2 = pthread_create(&input, NULL,
      &inputf, (void *) &data);
```

Implementation of threaded code and shared data

6 Socket Communication Between Processes

FlightGear allows for external control of program parameters through the implementation of sockets. Sockets for input and output are specified during the program call via command line arguments. The following represents the function call with arguments that we use to start FlightGear.

fgfs --generic=socket,out,10,127.0.0.1,5558,udp,FlightGearReceiver-Protocol --generic=socket,in,10,,6002,udp,test --nmea=socket,out,10,127.0.0.1,5557,udp --airport=ksfo --aircraft=c172p --in-air --altitude=1500 --vc=90 --heading=300 --timeofday=noon --prop:/controls/switches/starter=1 --prop:/instrumentation/attitude-indicator/config/tumble-flag=0

Output sockets are used to obtain telemetry information such as heading, airspeed, position and orientation. Input sockets allow other programs to control the aircraft. A third socket known as the NMEA socket relays the position of the aircraft through NMEA sentences. It simulates the output of a commercial GPS module.

6.1 Communication Protocol

Communication between FlightGear and the program use the UDP protocol instead of TCP. We chose the UDP socket protocol for our project for the following reasons:

a) No need for TCP error checking as both programs run on the same machine
b) Simpler and Faster

Data is sent from FlightGear at the rate of 10 Hz. Users have the option to specify which telemetry to receive through a configuration file placed in the FlightGear program directories. We decided to use the UAVPlayground config file as we were familiar with its functionality and knew it worked. Telemetry such as heading direction, air speed, etc is inputted from FlighGear to augment control algorithms. Refer to Appendix B for detailed operation procedures.

The input data needs to be parsed so that the telemetry can be used by the control algorithms. The parser for the input data is a state machine that is able to extract data by counting the number of ‘>’ characters which is used by the XML to specify data structures. The value is stored to a string buffer and then converted to a double using the atof function.

The output data is formatted according to the UAV Playground XML config file which specifies it to be four float values separated by tabs and delimitated by a newline. FlightGear takes in aileron, elevator, rudder and throttle as input commands. The following line produces the command string which is sent to FlightGear.

```
sprintf(buff, "%1.3f\t%1.3f\t%1.3f\t%1.3f\r\n", data_r->fg_out[0],data_r-
>fg_out[1],data_r->fg_out[2],data_r->fg_out[3]);
```

Professor Atkins provided us with the socket connection code (simple_sock) which allowed us to concentrate on the protocol issues right away. A slight modification was necessary to Professor Atkin’s code to meet our requirements. The buffer for output data and input data was either too large or too small. This was fixed by transitioning to a dynamic implementation so that data of varying lengths can be transmitted and received effectively.

Another issue we encountered was the length of the output command string changing as a result of negative values. As data is passed as a string and not binary negative values caused the length to change because of the ‘-‘ sign.

7 Data Logging and Data Sharing

A significant amount of data processing and data sharing is involved in our project. Data coming from FlightGear needs to be interpreted by the flight controller which, in turn, produces output that needs to be fed back to FlightGear. All of this takes place while loggers record all the input and output data to text files for post processing. The fact that each function runs on independent loops and different speeds necessitates a centralized and synchronized data sharing process. Data is shared via a common data structure that is passed to all threads. The structure contains arrays to hold input and output data as well flight modes which gets set by the User Input thread. We use mutexes to handle data access. This prevents read/write errors and data corruption that could be caused by multiple threads to access the same data points. There are three Mutexs that are used in the program

i) OutMutex – Protects Command data
ii) InMutex – Protects Telemetry data
iii) UIMutex – Protects Flight mode data

Every operation on the shared data structure, even checking of Boolean values, is protected by the mutexs.

As noted, all of the data is logged. This also allows us to import the telemetry and data into programs such as Matlab and Excel where we can analyze the performance of the PID controller gains. The user has the option to start or stop the logging at any point during the flight.

8 GPS Parsing and Recording

FlightGear is able to mimic GPS module output through the NMEA socket. The NMEA sentence format is a standard method of reporting GPS fix and GPS satellite information. The following sample line is one of the most commonly used sentences which reports time, latitude, longitude and altitude information.

*$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47*

The GPS parser is a state machine that checks for the GPGGA sequence then counts the commas to discern the locations of the various fields. These fields are stored in buffer of chars. Then using the function "atof()" the string value is converted to a double. The latitude and longitude information undergoes further processing as the minutes and degrees need to be converted to a decimal value. Our program uses the parsed data to generate a Google Earth kml file. The kml file is a specific type of XML file that is used to store a set of points to be displayed in Google Earth. The figure below, which shows our controller commanding a steady level turn, is generated by this parsing process.

Steady level turn executed by the aircraft

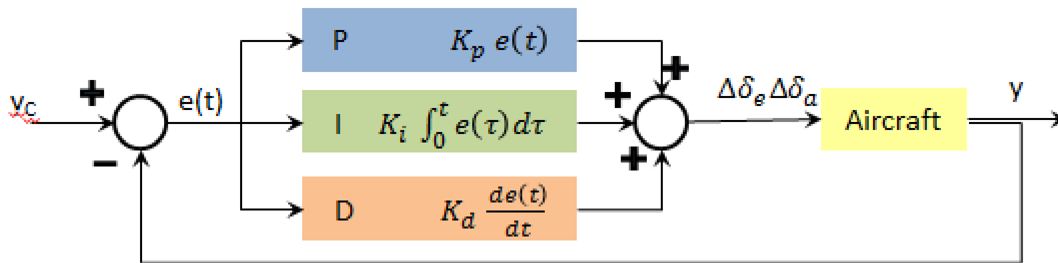
9 Terminal Input

Once the control software starts (fghead), the user is given the option to enter an input mode. There are four main input modes that are available to the user. The first, given by the 'l' command, is fly level. The program will stabilize the plane into a safe and level state. This is done by commanding both the roll and pitch to zero in the PID controller. This mode also takes no input value. The second mode, given by the 'a' command, takes in a desired altitude as the input value. The plane then commands the pitch to either a positive 10˚ pitch to increase altitude or a negative 10˚ pitch to decrease the altitude. When the current altitude is within 50 feet of the desired altitude, the program issues the fly level command by setting the input mode to 'l'. The third mode, given by the 'h' command, takes in a desired heading as the input value. The plane then determines whether it is shorter to turn to the right or to the left. The roll is then commanded to positive 20˚ or negative 20˚ to roll right or left, respectively. When the current heading is within 5˚ of the desired heading the program issues the fly level command similar to the change altitude mode. The last user command is the quit mode which is given by the 'q' command. This

command takes no input and simply turns the current controller off so that the plane can be manually controlled by the user.

10 Control Architecture

A PID controller is useful to maintain a set measurement without the need for an operator to constantly make adjustments. The controller used for this project is a basic PID controller as shown below. The program will take in a commanded value (y_c) and subtract the current value (y) from it to get the error (e(t)). In our case, the commanded value would be a 0˚ roll angle for the fly level mode with a possible current value equal to 20˚ if the plane were just coming out of a turning maneuver. The program will sample this current state at 10 Hz and feed it into the PID control function. The controller then sends this error value (e(t)) through the proportional, integral, and derivative gains. The tuning of these gains will be described later. The sum of these gains is then used to control the elevator ($\Delta\delta_e$) and ailerons ($\Delta\delta_a$). For our simple model, we are not currently using the rudder ($\Delta\delta_r$). The code, however, is written to be able to accept a rudder value if we choose to set it.

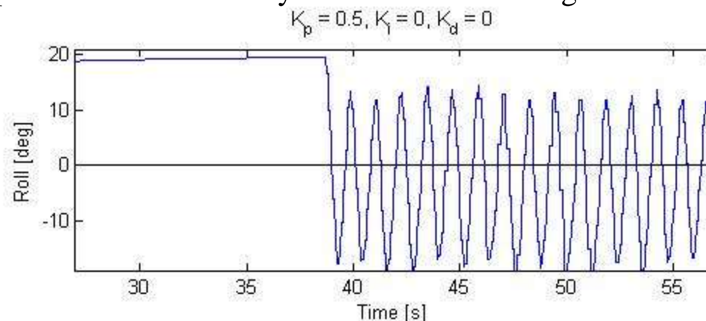


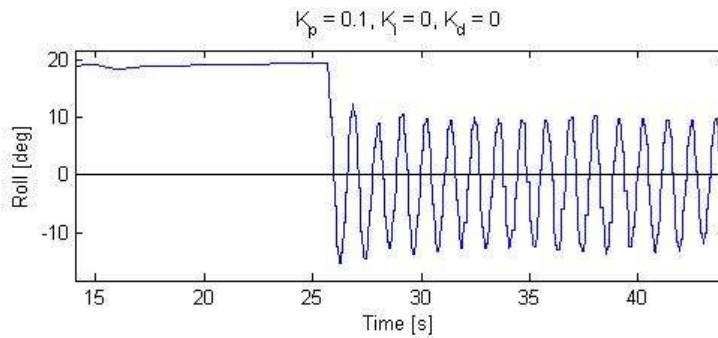
Idealize PID feedback controller

In order to tune the PID controller for each degree of freedom, we followed a basic procedure by starting with the proportional gain (P-gain) and then working to the derivative gain (D-gain), and finally the integral gain (I-gain). Throughout the process, previous gains were also slightly altered to create even better results. In the scope of this report, we demonstrate the procedure undertaken for tuning our roll PID gains, because the magnitude of pitch oscillations was less pronounced. The method was identical.

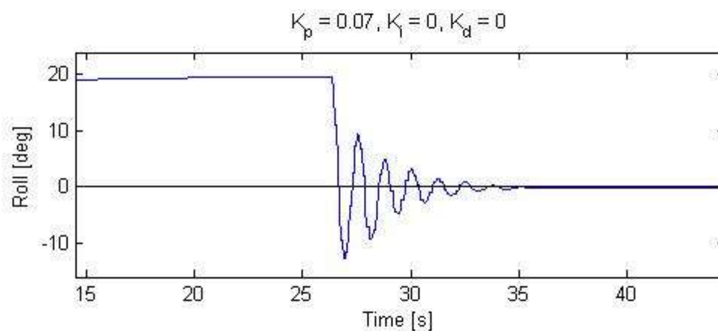
10.1 Stabilization

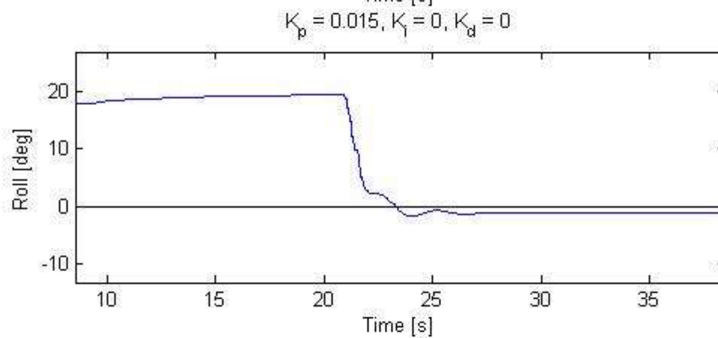
In order to see the full effect of the P-gain gain, we began by setting it to 0.5 then reduced it to 0.1 as shown below. The oscillations in these graphs are obviously undesirable. However, it is good to see that as the P-gain is decreased the amplitude of the oscillations is also decreased. We can continue to improve this controller by a reduction in the P-gain.



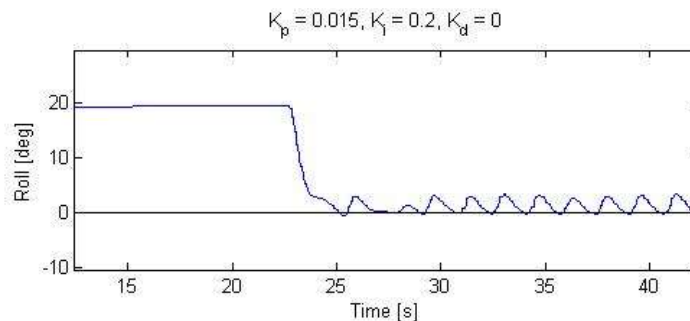


After a reduction of the P-gain to 0.07 the roll begins to stabilize after a few oscillations. Then, when the P-gain is reduced again to a value of 0.015, we get something closer to what we are looking for. The roll seems to be relatively stable; however, there is a noticeable steady-state error. This can be fixed with the addition of an I-gain.

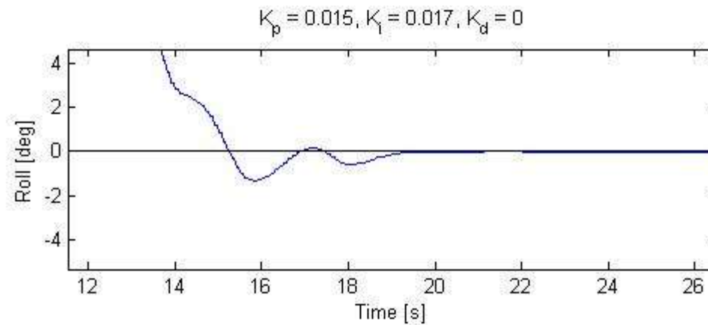


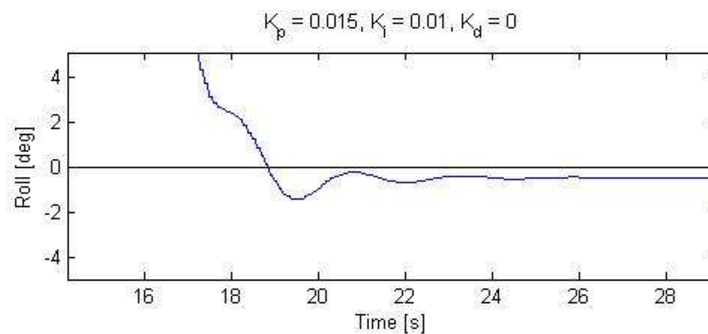


We started the I-gain at a high value of 0.2 to see what exactly it was doing to the roll. As shown below the I-gain has gotten rid of the steady-state error we saw before, but it also added a lot of oscillations. The addition of a D-gain can help to stabilize the roll, but first we want to create a better roll signal by reducing the I-gain.

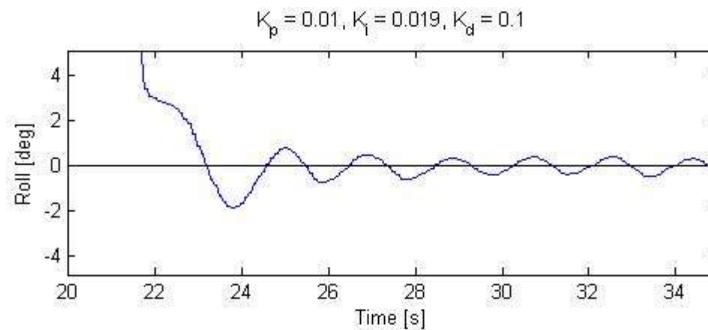


After a reduction in the I-gain to 0.017 we have a much better roll to work with when adding a D-gain. If the I-gain gets too low as in the case with I-gain = 0.01. We see that the purpose of adding the I-gain is not being fulfilled. The steady-state error is still visible. Therefore, we want an I-gain somewhere around 0.017 and greater-than 0.01.

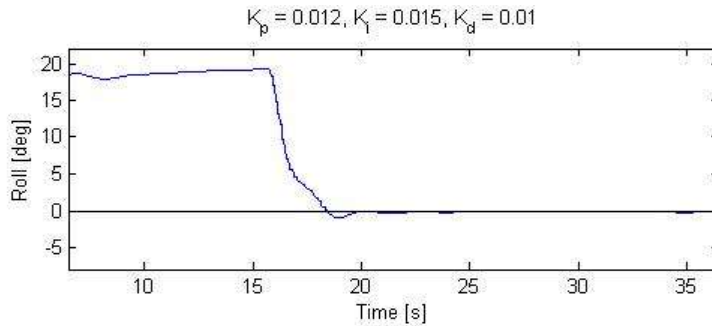




With the current P-gain set to about 0.015 and the current I-gain set to about 0.017, it is time to see what the addition of the D-gain will do for us. Again, we started off with a high D-gain of 0.1 in order to see what exactly it could do. This high D-gain caused the roll to become unstable so we continued to reduce the D-gain until this effect went away.



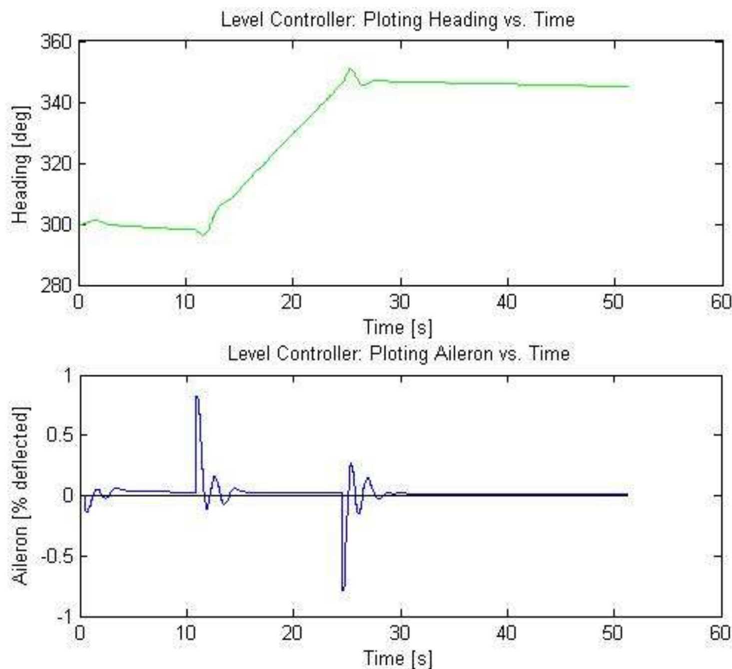
In conclusion, we chose a P-gain of 0.012, an I-gain of 0.015 and a D-gain of 0.01. These gains give the signal shown below. As shown, the roll moves quite quickly to the commanded value of 0˚ with one slight overshoot before stabilizing.



10.2 Steady Flight Maneuvers

With steady level flight (stabilization about 0 degrees) properly tuned, we were able to move forward with basic maneuvering. The one described here is a sample of a steady level turn, although our controller is able to perform an equivalent steady climb and the results are almost identical.

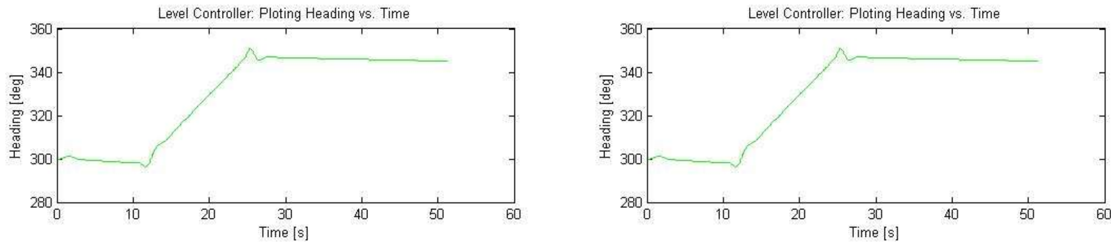
As an example, we initialized FlightGear to a heading of 300 degrees. After approximately 10 seconds, the user commanded a new heading of 345 degrees. The plane promptly banked right until reaching the desired heading—at which point it set the flight mode to 'l' for steady level flight. The following figure shows the recorded heading during flight and the subsequent one displays the commanded aileron position from the beginning to end of the turn.



Thus, these Matlab plots showing command and control outputs demonstrate the capabilities of our FlightGear communicator and controller. After de-scoping from advanced flight path and optimizing trajectories, *enabling* the basic control through robust communication became our primary objective, which we successfully delivered and demonstrated.

11 System Performance

The primary metrics in evaluating the performance of our testbed are repeatability and reliability. By delivering a pure software based testbed, we have enabled researchers the ability to replay and repeat specific control algorithms with greater precision or with incremental adjustments. We proved the repeatability of our system by hard coding a steady level turn of 345 degrees after 10 seconds of run time. This test was run twice from the same initial position (airport and altitude) and nearly the same time of day (offset by 1 minute, the duration of Trial 1). Note that realtime weather was enabled.



Trial 1 and Trial 2 are identical in a steady level turn

The errors between trials at any given time were less than the realistic precision in aircraft state (for heading—errors smaller than arc-seconds). Thus we concluded that our testbed was reliable and repeatable. Also, since it is open source and easily editable, the controller can be tuned, adjusted, or fully replaced quite easily and quickly.

Regarding measured processing time, our custom C++ architecture exhibited none of the memory leaks observed with UAV Playground. Unlike UAV Playground, our custom controller is multi-threaded so that transmit and receive periods can be appropriately coordinated. Worst case operating period is less of a concern for this testbed than it would be for an operational controller on an embedded system. However, it was important to stagger threads with *usleep* commands in order to limit the access to shared data. For example, our *control* thread can operate at a frequency greater than 100 Hz. Since our send and receive threads are only configured to run at 10 Hz, our controller would redundantly enable the mutexes 10 more times than required per cycle. Thus, we included enough sleep for the *control* thread to operate at approximately 20 Hz. This still ensures that we do not miss a telemetry reading.

Of course, our send and receive socket functions could update at much higher frequencies, especially on a shared machine. However, with UDP protocol, there is inherently greater risk in losing telemetry. We chose to imitate a real-time embedded system as closely as possible—many commercial GPS modules operate between 1 and 10 Hz which motivates our designed processing time. In the scope of this project, we have designed a testbed more than a specific controller. Although we did prove acceptable PID control, repeatability of the system was a bigger driver than runtime of a specific control algorithm.

12 Education Value

FlightGear was the ideal platform to test all of the skills developed through AERO 450, Flight Software Systems. As a team, we were able to devote all of our time to the software and let FightGear simulate the hardware for us. This allowed us to avoid sinking time into

malfunctioning hardware. During the project, we were able to gain experience working with both the Java and C++ languages.

We started with trying to debug Java then moved to C++ when the scope of the project was changed. We have also become familiar with using more code editing software such as Eclipse. Within the C++ language, we were able to look more closely at some of the advanced C++ programming taught in this course such as sockets (TCP/UDP) and multi-threaded code. Lastly, we have successfully merged our software knowledge with real-time aerospace control theory, implemented a PID controller to stabilize the aircraft, and logged and post-processed sensor and GPS data.

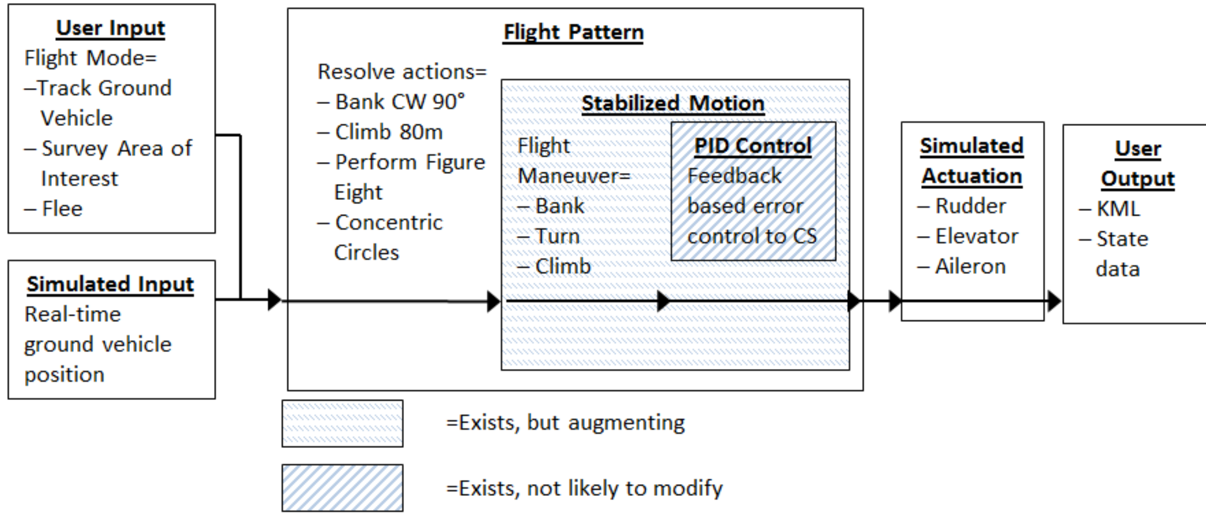
13 Broader Impacts

The past decade has seen the widespread introduction of free and open source software to the public domain. This movement refers to the freedom to copy and improve upon available source code. Such a peer-to-peer development strategy has advanced the scope of functional electronics, for the benefit of both individual and corporate growth. Over this semester, our team has used and built upon these readily accessible tools to efficiently begin work on this testbed.

The results of our work are expected to be delivered to the open source community at the conclusion of the semester. In the same way we are building on existing foundations, future researchers will be able to contribute to our work. Moreover, through the increasing globalization of shared data, this flexible testing platform can be distributed to academia and research labs across the globe.

An established autonomous vehicle testbed provides researchers with an intermediary step between software protocols and full scale reality. Such technology will accelerate the contributions to the emergent field of Unmanned Arial Vehicles.

Appendix A: Original Proposal Schematic



Appendix B: Procedure for Running our Final Project

All procedures are assumed that the user is running in a linux environment.

Copy the XML protocol files (FlightGearSender-Protocol.xml and FlightGearReceiver-Protocol.xml) into the FlightGear source folder (/usr/share/games/flightgear/)

Configure the fghead.sh batch file with the current directory (chmod +x fghead.sh). Run the FlightGear application by executing fghead.sh

Open a second terminal, wait approximately 10 seconds, and run fghead

Wait for the UDP sockets to open (a message will appear in the terminal) and choose desired flying mode. FlightGear will reflect the controller's operation.

Sample desktop setup