The *Flying Gator*: Towards Aerial Robotics in occam-π

Ian ARMSTRONG, Michael PIRRONE-BRUSSE, Anthony SMITH, and Matthew JADUD

Allegheny College
Meadville, Pennsylvania, 16335, USA

{armstri , pirronm, smitha7 , mjadud} @allegheny.edu

Abstract. The *Flying Gator* is an unmanned aerial vehicle developed to support investigations regarding concurrent and parallel control for robotic and embedded systems. During ten weeks in the summer of 2010, we designed, built, and tested an airframe, control electronics, and a concurrent firmware capable of sustaining autonomous level flight. Ultimately, we hope to have a robust, open source control system capable of supporting interesting research questions exploring concurrency in real time systems as well as current issues in sustainable agriculture.

Keywords. aerial robotics, occam-pi, Transterpreter, Arduino, ArduPilot

Introduction

Aerial robotics platforms provide authentic, real-time environments for testing concurrent control systems. While a ground based robot must always be on the lookout for navigational obstacles, it can almost always stop and "look around." Fixed-wing aerial platforms, on the other hand, cannot stop: they must constantly (and quickly) monitor flight parameters to maintain level flight or progress towards a goal.

The real-time challenge for unmanned aerial vehicles (UAVs) is compounded by the large number of sensors and actuators involved in flight. A typical inertial measurement unit (IMU) might include an accelerometer and gyroscope (for tracking an aircraft's attitude in three-dimensional space), a magnetometer (to serve as a compass), an airspeed sensor, and a GPS (for position and altitude). In addition to sensing, at least one motor and three control surfaces (for fixed-wing UAVs) must be managed as outputs. In between these inputs and outputs are complex filtering and control algorithms that turn noisy sensor inputs into useful information and guide the aircraft on its flight path.

In this paper we present the *Flying Gator*, an aerial robotics platform capable of sustained level flight with a control system developed in occam-π [1]. We begin by discussing the construction and design of the UAV, the sensing challenges in maintaining flight and how process-oriented software design helped manage these challenges. We then compare our work (briefly) to the ArduPilot, a popular, open-source UAV autopilot developed in C++ for the same control hardware, and close with a brief discussion of next steps in this line of work.

1. The Flying Gator UAV

The *Flying Gator* is a step in an ongoing investigation regarding concurrent and parallel control for robotic and embedded systems [2,3]. Our goal is for the *Flying Gator* to support collaborative research in environmental science, serving as a platform for low-cost aerial

sensing and imaging. During ten weeks in the summer of 2010, we took our first steps down this path, designing, building, and testing an airframe, electronics, and a concurrent control system capable of sustaining level flight.

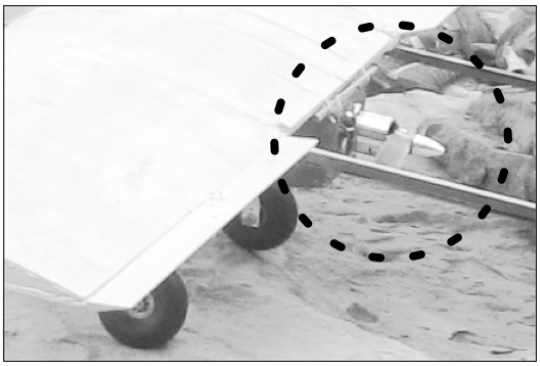
1.1. Airframe

The *Flying Gator*'s airframe satisfies a challenging range of requirements. Foremost, the aircraft has a strong and lightweight airframe, thus maximising the *Flying Gator*'s potential payload (how much it can carry beyond essential operating components). Because the aircraft is ultimately intended to support aerial sensing, a mount was included for either a still or video camera forward on the fuselage, in addition to the control electronics and batteries used to power the UAV. As a research platform, the *Flying Gator*'s large wingspan leads to more stable flight characteristics at low speeds, and better control in windy or gusty weather.

(a) Spar-and-rib wing construction.

(b) The *Gator*'s fuselage construction.

(c) The *Gator*, complete and painted.



(d) A pusher prop design.

Figure 1. The construction of the *Flying Gator*.

Many materials were considered while a strong, durable and lightweight airframe was being sought after. Foam and balsa were layered throughout the build processes for the wings and fuselage. Foam is a light and flexible materials, and is easy to use when building a low-weight UAV. Using strategic wooden reinforcements and layers of thin balsa (still very light, but more ridged than foam), an airframe resistant to the stresses of flight was completed.

Traditional spar-and-rib building techniques were used for the left and right wing panels (Figure 1(a)). Central wooden spars were used in place of foam to allow for maximum strength and rigidity. The fuselage was built in a similar manner, built up as a foam box with balsa reinforcements to support the nose gear, main gear, and motor mounts (Figure 1(b)). Further reinforcement was added to the nose of the fuselage after flight testing indicated that landing stresses might damage the aircraft. Wooden tail booms, reinforced with carbon fiber, provided rigidity and were easily removed for transport.

The fuselage must support a motor, a camera, and provide mounting points and protection for control electronics. The *Flying Gator* uses a "pusher prop" design (Figure 1(d)), which provides an unimpeded view out of the front of the aircraft as well as increased airflow over the control surfaces on the aircraft (leading to better reaction times). The camera mount not only has an unimpeded view from the nose of the aircraft, but it also provides critical weight forward of the wing to counterbalance the motor. The complete *Flying Gator* is pictured in Figure 1(c), and video of radio controlled[1] and autonomous flights[2] are available online.

1.2. Electronics

It was paramount that the design of our electronics system should support the operation of the *Flying Gator UAV* in a manner that is safe and within regulations laid out by the United States Federal Aviation Authority[3]. The guidelines can be summarised as:

1. Stay below 400' (120m).
2. Stay away from built-up areas.
3. Maintain "pilot in command."

Guideline number 3 means that a human operator must always be able to take control of the aircraft at any time. Our design must, therefore, accommodate both a traditional radio control system as well as our electronics for autonomous control—and provide a means for switching between them.

The aircraft flight electronics such as motors and servos were typical radio control aircraft components. A 6-channel, 2.4 GHz Spektrum radio system was used for manual ground control, Hitec servos were employed on all of the control surfaces, and Turnigy components were used in the power system. Although the *Flying Gator* has a large airframe, our choice of a 43mm diameter brushless electric motor provides ample power to sustain flight.

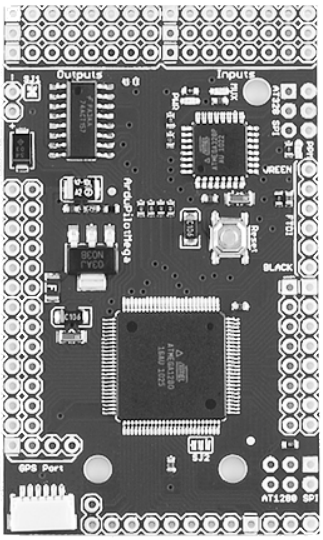


Figure 2.: The ArduPilot Mega.

Rather than developing custom control and switching electronics, as these constitute critical safety systems in a UAV, well-tested, open hardware solution was chosen: the ArduPilot Mega (Figure 2) [4]. This $60, 4cm x 7cm board is built around the Atmel ATmega1280 processor. The ATmega1280 runs at 16MHz, has 128KB of flash for code and 8KB of RAM for executing programs, and 100 pins for interacting with the world. Of these pins, the ArduPilot Mega exposes sixteen 10-bit analog-to-digital channels (critical for reading sensors) and ample pulse width modulation hardware for precisely driving servo outputs. Most importantly, the board includes a separate ATmega328 (a smaller processor in the same family) that comes pre-configured with firmware to handle switching between an autonomous mode (on the ATmega1280) and inputs from the ground control radio.

Our sensor input board was custom designed. We chose to use the 6DOF Razor IMU from SparkFun Electronics [5], which incorporated the LPR530AL and LY530ALH gyroscopes (pitch/roll and yaw, respectively) from ST Microelectronics, and the ADXL335 three-

[1]Radio controlled flight: `http://goo.gl/g4rfa`

[2]Autonomous flight: `http://goo.gl/hFa6C`

[3]FAA recommendations and guidelines: `http://goo.gl/EtGoz` and `http://goo.gl/kv7Xy`

axis accelerometer from Analog Devices. The gyroscopes are capable of registering a change of up to 300 degrees in a second, and the accelerometers are sensitive within the range of -3 to +3 gravities. Our GPS receiver (which was prototyped but not used in flight testing) used the released Venus chipset, which typically provides better than 2.5m accuracy [6].

2. Sensing Challenges

Keeping a small aircraft in the air requires information regarding the attitude of the plane that is constantly updated and, generally speaking, accurate. To maintain level flight, the *Flying Gator* uses accelerometers and gyroscopes, the combination of which is referred to as an inertial measurement unit, or IMU.[4] Accelerometers use the acceleration due to gravity to help us calculate the direction of "down," and gyroscopes measure the rotational acceleration around the x-, y-, and z-axes. By combining these two sets of data into one angle measurement per axis, we can accurately calculate the plane's attitude with respect to the horizon.

A major challenge with the sensors on an IMU is that they are exposed to a great deal of vibration as a matter of course. The motor, which is directly attached to the airframe, is one significant source of vibration; external conditions (gusts of wind and turbulence) are another. Both the accelerometers and gyroscopes in the IMU are negatively impacted by the environment in differing ways. Vibration, external forces (such as from turns), and time all play a role in decreasing the accuracy of the sensor data being reported.

2.1. Vibration

The main source of vibration on an aircraft is from the power system. Even a well-isolated and well-balanced electric motor is a significant source of airframe vibration. Vibrations will always occur due to efficiency losses in the propeller, as well as issues surrounding balance: a small motor spinning at 8000RPM, mounted to a foam-and-balsa frame, is never going to be perfectly balanced. These vibrations affect both accelerometers and gyroscopes, and cause variations in sensor data that must be managed by our controller.

2.2. External Forces

Ideally, our accelerometer would measure only the force of gravity, or "down." Unfortunately, this is not how an accelerometer works in flight: centripetal forces (generated when the aircraft is moving in a circular path, or turning) cause accelerometers to read inaccurately because they are no longer measuring just gravity. With this extra force, it is not possible to determine our orientation in relation to the ground solely based on the accelerometer. Gyroscopes are not typically affected by external forces (but are affected by vibration) as they only measure *instantaneous* acceleration. For this reason, we use a combination of sensors to augment and correct one another's observation of the aircraft's orientation.

2.3. Sampling Frequency

The *Flying Gator* is easily capable of speeds in excess of 5 meters/second (18 KPH). Our typical flight altitude is on the order of 25 meters, meaning (ignoring gravity) that the aircraft is capable of diving into the ground in less than five seconds. For this reason, a small aircraft should sample its IMU at least 60 times per second, and then update control surfaces in response to changes in position, attitude, and speed. Our initial test flights focused on level

[4]Our initial goal was to achieve level flight, and therefore a GPS, which provides altitude and heading, was not employed in our initial design.

flight at a fixed speed, meaning we concerned ourselves only with processing gyroscope and accelerometer data.

A gyroscope provides an instantaneous reading regarding the rate of rotation around an axis. We must therefore read the sensor often and integrate to develop a meaningful gyroscope reading regarding our attitude in 3-dimensional space. Unfortunately, gyroscopes are prone to drift; in addition to having to read them continuously, we must also combine them with accelerometer data to correct for their drift, which increases the number of sensors we must monitor in our IMU.

2.4. Integer vs. Floating Point

The ATmega1280 has no floating point unit; all trigonometric and floating point operations must therefore be emulated on this processor. The ArduPilot codebase uses floating point mathematics, but can get away with it, because it is written in C++, which is executed natively. When running occam-π programs on the Transterpreter, we take two performance hits. Our initial implementation included trigonometric functions and floating point values for processing our accelerometer values, which meant that we had the overhead of 32-bit floating point emulation in both the virtual machine and the hardware. In developing processes to handle sensor data, we began by using the occam-π `FLOAT` type, which (on the Arduino) is implemented as a 32-bit value across two 16-bit registers. We were able to process our x-, y-, and z-axes for the accelerometers and gyroscopes approximately four times per second—this was far too slow.

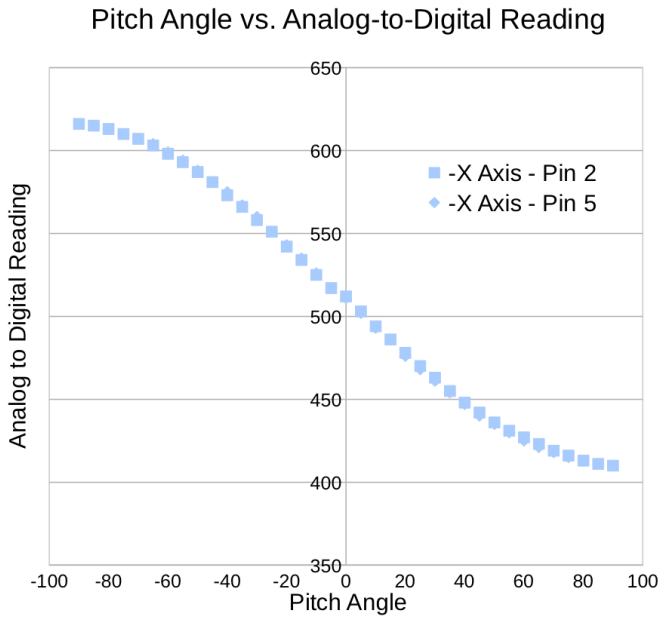


Figure 3. Averaged accelerometer readings on x-axis, ADC inputs 2 and 5.

We realised, however, that (1) we did not need the level of precision that a `FLOAT` provided, and (2) that we could linearly map the raw analog-to-digital (ADC) readings from our accelerometers to an orientation in degrees. To generate this map, we built a goniometer[5] from two protractors, a cut down wooden ruler, and some tape. We mounted our IMU in our goniometer, and averaged 30 ADC readings from -90 degrees to 90 degrees in 5-degree increments on two different pins (Figure 3). Then, using linear approximations, the ADC values could be quickly converted directly into angle values[6]. This improvement made our

[5] A goniometer is a tool for rotating an object to a specific angular position.

[6] Full source available at `http://goo.gl/Qn1x2`.

IMU-processing run approximately 40 times faster, which was fast enough for us to develop control algorithms that would allow the *Flying Gator* to fly autonomously.

3. Flight and Control

Our implementation of a parallel control system for the *Flying Gator UAV* was a "clean room" implementation with respect to the ArduPilot source code [7]. We did not study it extensively or attempt to translate some/all of this code from C++ into occam-π. The ArduPilot firmware is implemented around a single run-loop (no multithreading) and hardware interrupts, and more than 50 global variables.

This is not uncommon in the world of embedded systems. However, this "pattern" for software development in the embedded space is dangerous for several reasons, some of which can be avoided when using a process-oriented language like occam-π.

Safety in communication. The ArduPilot firmware uses at least 50 global variables that represent the complete state of the aircraft. These variables are essential to an aircraft's correct operation and, therefore, flight. It is not possible (by inspection) to say what functions in the control loop modify any or all of these variables without reading through each of them carefully, and it is possible that some future updates to the firmware will lead to a race hazard or perhaps a crashing condition in the firmware.

In occam-π, there is no global state: every single process communicates values safely over managed *channels*. Given that we are currently only developing our firmware using the core of occam-π (or, occam2.1), we know it is impossible for us to have global variables or encounter any of the basic race hazards when communicating values from one process to another over a channel.

One vs. many loops. There is only one loop in the entire ArduPilot control system. This means that no one part of the loop (e.g. function called from the loop) should take "too long," or it will slow the speed at which sensors are read, data is processed, and control surfaces are updated.

Any firmware developed in occam-π will typically have many dozens of processes running concurrently, yielding to each other cooperatively. Each occam-π process is likely executing on a continuous basis, and could be constructed to operate when other processes are not "busy." We design each process to have one task, so that if we discover an error in the operation of our control system, we can isolate the source of the error quickly. Further, as multicore processors become more common, we are confident that our firmware can be scaled to take advantage of this additional computational power to good effect.

Terminal behaviour. As we developed our firmware, working with a virtual machine was a massive boon. When crashing bugs were encountered, the Transterpreter virtual machine would report the line number where the error occurred. This alone saved countless hours of debugging time (compared to developing a similar system in C++), as at no point was a debugger (like `gdb`) required.

While occam-π provides a great deal of safety when authoring a concurrent runtime, it also provides tools for constructing scalable, complex systems. This leads to control systems that look more like networks of self-contained processes that communicate and collaborate in a way that sequential programs typically do not.

4. Architecture and Control

Our control system leverages the process-oriented nature of the programming language occam-π. The "network" of processes executing on the ArduPilot Mega concurrently reads from sensors, filters the data produced, makes decisions based on that data, and then drives small motors (servos) that alter the control surfaces of our aircraft. We begin with these small components, look how they fit into the overall picture, and close with a short discussion of the challenges of converting noisy sensor data into useful information.

4.1. Composing a Control System

The Arduino project has a library of code based on *Wiring* [8], which provides high-level convenience functions for interacting with the hardware. For example, to "turn on" digital pin 13 on the Arduino we would write `digitalWrite(13, HIGH)`. We contrast this with the kind of code that is written for many microcontroller projects: bitwise twiddling of registers that seems very complex to the novice programmer.

When programming the ArduPilot Mega in occam-π, we use a different library: *Plumbing*[7]. This library provides high-level abstractions for many common interactions with the microcontroller. In some cases, it is very similar to *Wiring*: `digitalWrite(13, HIGH)` from *Wiring* becomes `digital.write(13, HIGH)` in *Plumbing*.

While some parts of the library are very similar, other parts of the library provide a very process-oriented interface to the hardware. The `servo()` process is one example. A servo is a small motor that is used for moving the ailerons, elevators and rudders[8] of a small aircraft. To drive it, a series of precisely timed pulses are sent down the wire, and the rate at which the pulses are sent determines the degree of servo rotation. In the *Plumbing* library, the `servo()` process provides a single channel of type `SERVO`, which is defined by an occam-π channel protocol[9] (Listing 1).

```
PROTOCOL SERVO
  CASE
    max ; INT
    min ; INT
    div ; INT
    pos ; INT
    usec ; INT
    frequency ; INT
    start
    stop
:
```

Listing 1: The `SERVO` protocol.

To initialise a servo, values for `minimum`, `maximum`, and the number of `divisions` are transmitted to the process; these set the limits of the servo rotation as well as the number of divisions mapped into that range. In typical usage, values of 0, 180, and 180 provide access to the full range of the servo in one degree increments. Once these values have been set, the `start` message is sent to the `servo()` process, and it begins driving the servo. The programmer can then send a series of `pos` messages, which will cause the servo to rotate to and hold whatever position is communicated.

[7] Book and code available online at `http://concurrency.cc/`.

[8] The control surfaces of aircraft, used for effecting roll(rotation around the fuselage), pitch(up-and-down) and yaw(side-to-side), respectively.

[9] To save space, all of the documentation has been removed from this protocol; we recommend viewing the full source online—`http://goo.gl/yiiDl`.

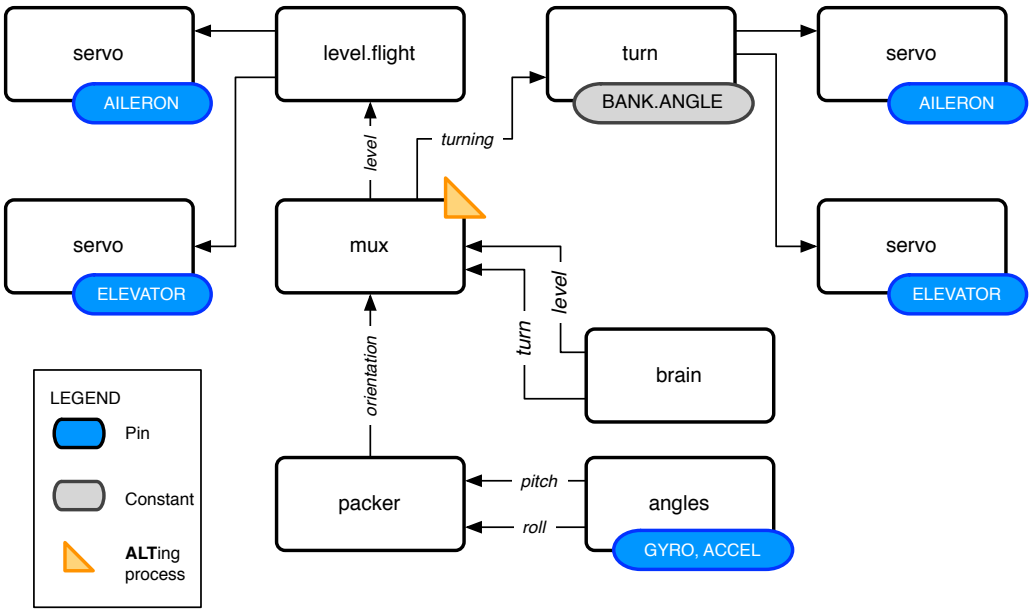


Figure 4. The process network executing on the *Flying Gator*.

4.2. The Process Network

We highlight `servo()` as an example of a process-oriented interface to hardware because it represents an encapsulation of state that is not present in the ArduPilot firmware. In contrast to the sequential approach, our firmware encapsulates this state in a network of communicating processes (Figure 4).

Given that our initial flight tests were focused on achieving level flight, our `brain()` process is very simple at the moment: it alternates between signaling for the start of level flight and a turn, leading to a rounded-rectangle flight pattern. The `angles()` process reads our sensors and filters the data, ultimately converting it into an orientation in 3-dimensional space (Section 4.3). Our multiplexer process watches the channels coming from the `brain()` and `angles()` processes and decides whether to signal the `level.flight()` or `turn()` processes, which in turn actuate the servos.

Where the ArduPilot software software uses global variables that can modify the state of the aircraft from anywhere at any time, we are guaranteed by our `mux()` process that only one of `turn()` or `level.flight()` will be executing at any given time. Therefore, only one process—responsible for turning or flying in a straight line—will be in control of the hardware. They are mutually exclusive processes, and their exclusion is guaranteed by the construction of our network.

4.3. Filtering

Clean and reliable sensor data is an ideal in the smooth and efficient operation of any aerial platform. Unfortunately, as we have seen, there are myriad sources of error and inconsistency that can effect the readings received from our sensors. Methods of data filtration that sanitise the raw sensor readings into workable values can range from the simple to the complex. The simplest way to filter noisy data is to average sensor readings over time. This approach may work in some instances, but it is not based on a physical model and may let unreliable data or outliers unduly influence our control choices. At the other end of the complexity scale is Kalman filtering, which can become both complex in its implementation and computationally expensive [9], especially on a resource constrained device like the ATmega1280. The *Flying*

Gator uses complementary filtering, a robust method that is somewhere in-between these two extremes.

A complementary filter takes input from the accelerometers and gyroscopes and combines the raw sensor readings to produce a more accurate picture of the aircraft's orientation in space. The filter can weight the inputs by positively or negatively biasing them. We might do this to take into account a sensor that is inaccurate in some systematic way. For example, we might weight gyroscopes more heavily during turns, as the accelerometers will be affected by centripetal forces experienced during the turn.

```
1  PROC comp.filter (CHAN INT gyro?, accelerometer?,
2                           theta!, VAL INT gain, dt)
3   INT gyr, accel:
4   INITIAL INT angle IS 0:
5   WHILE TRUE
6     SEQ
7      gyro ? gyr
8      accelerometer ? accel
9      angle := (((gain * (angle + (gyr * dt))) +
10               ((100 - gain) * accel)) / 100)
11     theta ! angle
12  :
```

Listing 2: The *Flying Gator*'s complementary filter.

Listing 2 is our implementation[10] of a complementary filter for the *Flying Gator* [10]. First, the processed gyroscope and accelerometer values are read into their respective variables on lines 7 and 8. On line 9 the filtering takes place. The "gain" parameter in our filtering process allows us to trim the balance between the gyroscope and accelerometer readings: a value of 50 would mean that each is given equal weight in our calculation. Higher values bias the filtering process towards the gyroscope, and lower values take the accelerometer into account more heavily. (The *Flying Gator* typically flew with a value of 80, biasing heavily towards the gyroscopes.) We replicate this process for both the pitch and roll of our aircraft (the x- and y-axes). The last line of this `PROC` ships the filtered angle out to wherever it needs to be used.

Along with our complementary filter we also applied a modified Runge-Kutta method to smooth out the gyroscope data. This implementation[11] looks at past values and averages out large, sudden, inconsistent changes in the incoming data.

Listing 3 is our implementation of a Runge-Kutta for pre-processing gyroscope data before passing it to the complementary filter. Lines 5, 6, and 7 are where most of the work happens. We take a weighted average of the past three averaged readings, and then combine that value with the current reading. This eliminates any large inconsistencies in the gyroscope data, as it is prone (in flight) to be noisy—but consistent, extreme changes (e.g. a sudden dive due to a gust of wind) is still passed through to the rest of the control system.

5. Maintaining Flight

Knowing the limitations of our hardware, we attempted to keep our flight control processes as simple as possible. Based on our prior experience with small aircraft, we recognised that

[10] Full source available at `http://goo.gl/etVj9`.

[11] Full source available at `http://goo.gl/YISTx`.

```
1  PROC gyro.filter (CHAN INT input?, rate!)
2    SEQ
3      ...  initialisation
4      WHILE TRUE
5        SEQ
6          current.rate := ((current.rate) +
7                           ((((filter.arr[1] + filter.arr[3]) +
8                           (filter.arr[2] * 2)) / 4))) / 2
9          rate ! current.rate
10         -- Cycles old values through temporary variables
11         -- and holding the last four values
12         filter.arr [0] := filter.arr [1]
13         filter.arr [1] := filter.arr [2]
14         filter.arr [2] := filter.arr [3]
15         filter.arr [3] := current.rate
16         input ? current.rate
17 :
```

Listing 3: Applying a modified Runge-Kutta to the gyroscope data.

there is an inverse relationship between the attitude of the plane and the servo movements required to bring the plane back to level flight. Our implementation[12] of level flight consists of five lines of code (Listing 4).

```
1  PROC level.flight (CHAN IMU.DATA imu?, CHAN SERVO s, s2)
2    INT  servo.pos.pitch, servo.pos.roll:
3    IMU.DATA pos:
4    WHILE TRUE
5      SEQ
6        imu ? pos
7        servo.pos.pitch := (pos[pitch] *
8                           ((-1) * PITCH.SERVO.MULTIPLIER)) + 90
9        s ! pos ; servo.pos.pitch
10
11       servo.pos.roll  := (pos[roll] *
12                          ((-1) * ROLL.SERVO.MULTIPLIER)) + 90
13       s2 ! pos ; servo.pos.roll
14 :
```

Listing 4: The level flight "reflex."

After reading in the filtered data from the accelerometers and gyroscopes on the `imu` channel, we look at both the pitch and roll separately. We take the inverse of the angle value from the `imu` and multiply it by the appropriate servo multiplier. This serves as a tunable scaling factor for controlling how much the servo moves in response to our sensor data. We then add 90 degrees to account for the particular orientation of the servos in the *Flying Gator*.

The `level.flight()` process serves two purposes in our code. First, it is a simple mapping of data to control that fits naturally within a process-oriented paradigm. Second, and more importantly, it reflects how we wish to begin thinking about future iterations of our control system. Specifically, we have begun thinking about what flight might look like in terms of Brook's subsumptive control paradigm [11]. Inspired by earlier work regarding

[12]Full source available at `http://goo.gl/C2bYx`.

subsumption and occam-π, we imagine that processes like `level.flight()` might serve as low-level behaviours that ultimately would (if necessary) subsume more complex behaviours like waypoint navigation and path planning [12].

6. Future Work

The *Flying Gator* serves both as a platform for ongoing case studies regarding the application of process-oriented programming to real-time robotic control as well as a research platform in its own right. Ultimately, we hope to have a robust, open source control system capable of supporting interesting research questions exploring current issues in sustainable agriculture. Our interest in developing behavioural and hybrid control architectures moves us towards this goal, as will several important, short-term goals:

Ground communications. The *Flying Gator* has no way of communicating with the controllers on the ground. Given the low cost of low-power radio systems, it would make sense to integrate a way for the ground controller and UAV to communicate—perhaps leveraging a protocol like MAVLink, which is designed for this purpose [13].

Logging. In our initial designs, we did not integrate logging. Given that it is possible to interface to gigabytes of storage on microSD cards very inexpensively from the ArduPilot Mega, we will include a "black box recorder" in future revisions of our hardware and software, for both analysis and debugging purposes.

Additional aircraft. Our entire project was executed in 10 weeks, from design, to development, and testing. During the 9th week of our project—after two weeks of test flights—the *Flying Gator* crashed. We claim that our code was not at fault, as the tail *physically broke* while cruising at 25m altitude. Our control electronics survived, as did the motor (due, in large part, to the *Flying Gator*'s pusher-prop design)... but our foam-and-balsa fuselage and wings did not. Additional aircraft would both open up the doors to exploring multiple agent systems as well as allow for research to continue in the event that our aircraft experiences a sudden and unexpected loss of altitude.

Simulation frameworks. Simulators are valuable for multiple reasons. Inclement weather and unexpected (total) aircraft failure can slow ongoing research and development. Simulators allow for testing in extreme conditions as well as on a wide variety of aircraft—something that is hard to do in the real world.

7. Conclusion

The *Flying Gator* was designed, developed, and tested over a 10-week period during the summer of 2010. Our control system, implemented in the programming language occam-π, allowed us to rapidly explore ideas regarding the efficient control of a small aircraft using the ArduPilot Mega, a low-cost piece of open hardware designed for this purpose. Although our aircraft experienced airframe failure at the end of the summer, we consider the work a success, and intend to continue our explorations regarding concurrent real-time control of autonomous aircraft when we complete construction of the *Flying Gator II*.

Acknowledgements

The *Flying Gator* project was supported in part by the Shanbrom Fund and the Department of Computer Science at Allegheny College, and a grant from the Institute for Personal Robotics (`http://www.roboteducation.org/`). Figure 2 used with permission from SparkFun Electronics.

References

[1] Peter H. Welch and Fred R. M. Barnes. Communicating Mobile Processes: introducing occam-π. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[2] Matthew C. Jadud, Christian L. Jacobsen, and Jonathan Simpson. Patterns for programming in parallel, pedagogically. In *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pages 231–235, New York, February 2008. ACM Press.

[3] Matthew Jadud, Christian L. Jacobsen, Jon Simpson, and Carl G. Ritson. Safe parallelism for behavioral control. In *2008 IEEE Conference on Technologies for Practical Robot Applications*, pages 137–142. IEEE, November 2008.

[4] Chris Anderson and Jordi Munoz. The ArduPilot Mega. `http://diydrones.com/profiles/blogs/ardupilot-mega-home-page`, February 2011.

[5] SparkFun Electronics. IMU Analog Combo Board Razor – 6DOF Ultra-Thin IMU. `http://www.sparkfun.com/products/10010`, February 2011.

[6] SparkFun Electronics. Venus GPS with SMA Connector. `http://www.sparkfun.com/products/9133`, February 2011.

[7] Jordi Munoz and Jason Short. The ArduPilot source. `http://code.google.com/p/ardupilot/source/checkout`, February 2011.

[8] Hernando Barragán. Wiring. `http://wiring.org.co/`.

[9] Jeffrey A. Kramer. Accurate localization given uncertain sensors. Master's thesis, University of South Florida, 2010.

[10] Shane Colton. The Balance Filter: A Simple Solution for Integrating Accelerometer and Gyroscope Measurements for a Balancing Platform. `http://web.mit.edu/scolton/www/filter.pdf`, February 2011.

[11] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[12] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile Robot Control - The Subsumption Architecture and occam-π. In P. Welch, J. Kerridge, and F. Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering*, pages 225–236, Amsterdam, September 2006. IOS Press.

[13] Lorenz Meier. MAVLink Micro Air Vehicle Message Marshalling Library. `http://qgroundcontrol.org/mavlink/start`, February 2011.