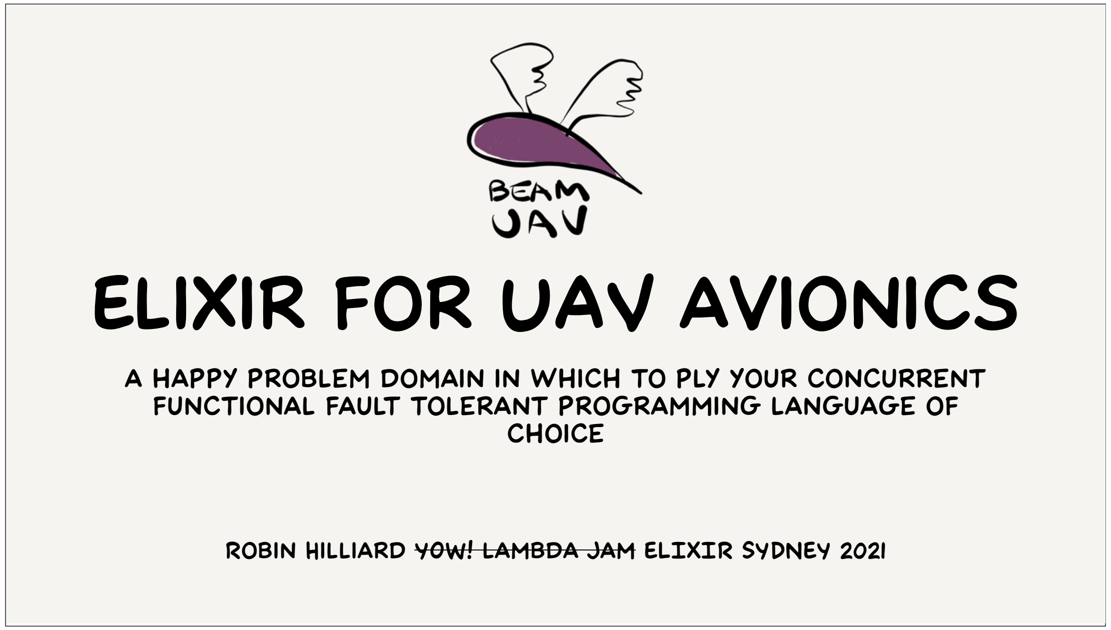


BEAM UAV

ELIXIR FOR UAV AVIONICS

A HAPPY PROBLEM DOMAIN IN WHICH TO PLY YOUR CONCURRENT FUNCTIONAL FAULT TOLERANT PROGRAMMING LANGUAGE OF CHOICE

ROBIN HILLIARD ~~YOW! LAMBDA JAM~~ ELIXIR SYDNEY 2021

Hi I'm Robin. By day I'm the Founder and CTO of RocketBoots, a company working on computer vision products for banking and retail clients. In the evenings and weekends one of my hobbies is programming, building and simulating UAVs or Uncrewed Aerial Vehicles, otherwise referred to as drones.

This has particularly been the case since 2013, when my friend Brian and I prepared a UAV for the 2014 UAV Outback Rescue Challenge. This competition, started in 2007 by the CSIRO and QUT, attracts teams from around the world to compete in challenges such as dropping a bottle of water for a lost "bush walker" returning a blood sample from a sick farmer or providing video and audio feeds of an accident victim to an approaching a medical team.

Uniquely the competition takes place beyond line of sight at distances of tens of kilometres from the launch point under a special dispensation from the Civil Aviation Safety Authority. This is illegal in most countries where people are well off enough to be involved in UAVs as a hobby, so it's a great opportunity for all involved.

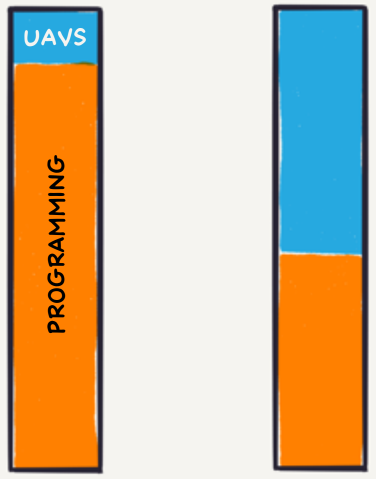
It is a huge amount of work to put together an entry for this competition. I've had three attempts at getting to the competition so far. Our team Beam UAV entered the 2020 competition which is now deferred till 2022 at the earliest due to COVID. In the process I've built and bent a few airplanes, learned a lot, written many lines of code and had a lot of fun. My assumption is that some of you are interested, if not in the UAV Challenge, then at least in building, programming and flying UAVs.

ASSUMED KNOWLEDGE

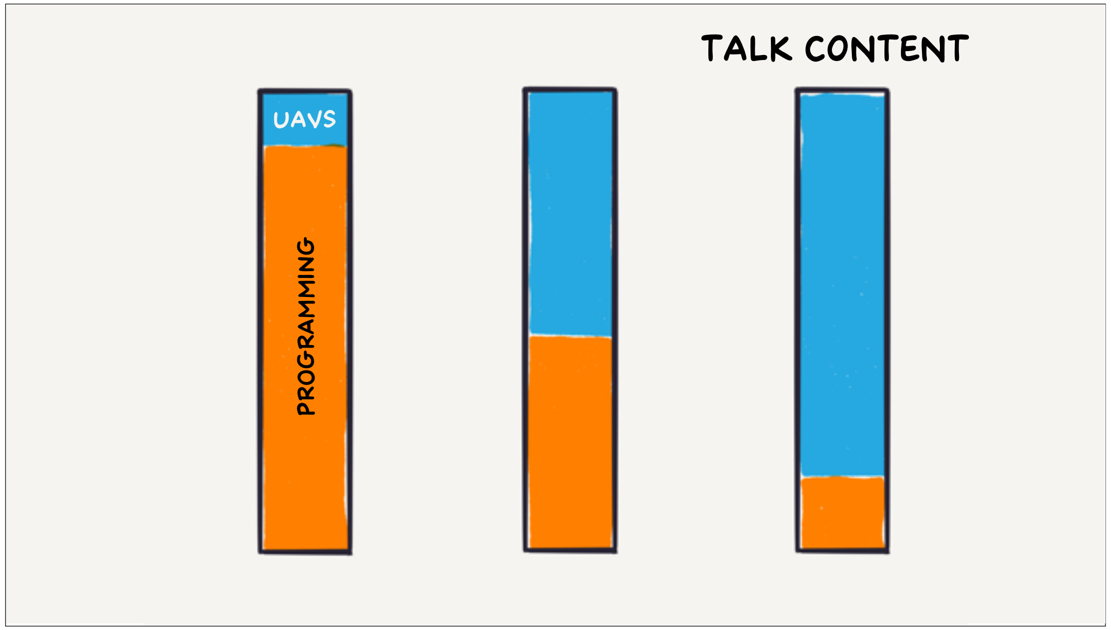
UAVS
PROGRAMMING

This is a programming conference so I assume you know a lot about programming and less about UAVs.

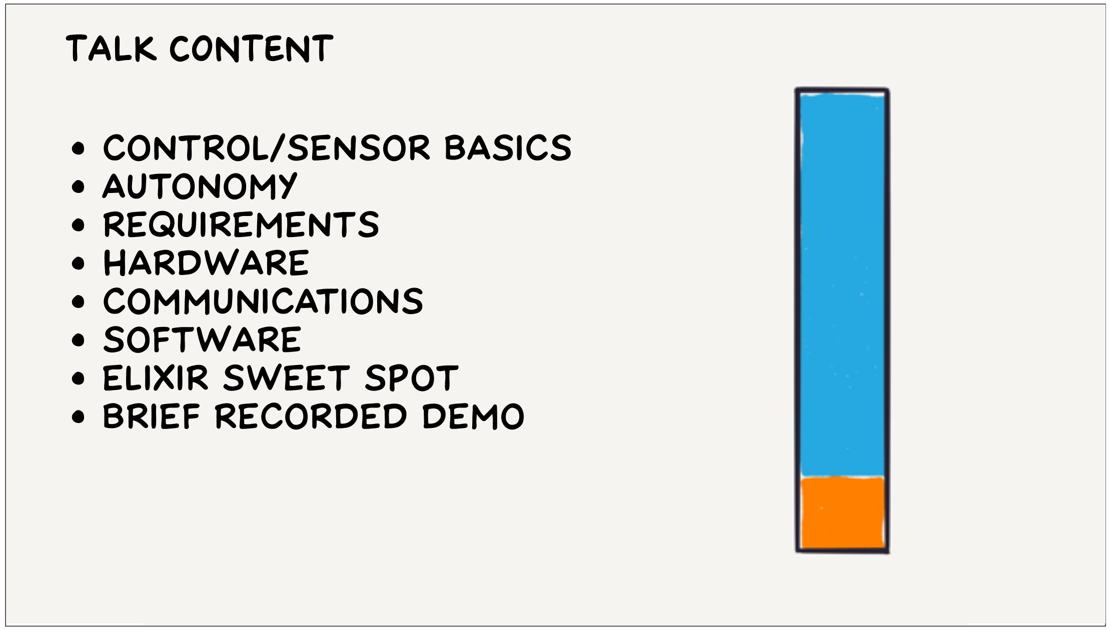
POST TALK TARGET



At the end of the talk I want to equip you with enough knowledge about UAVs to be able go off and try something out.



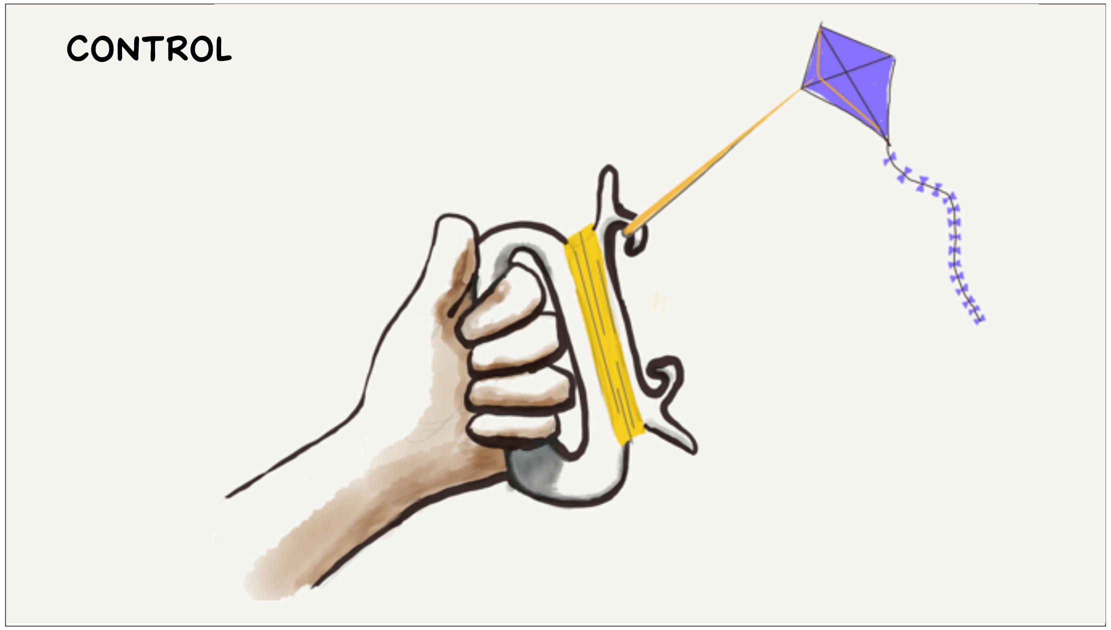
So this talk is mostly going to be about UAV topics.



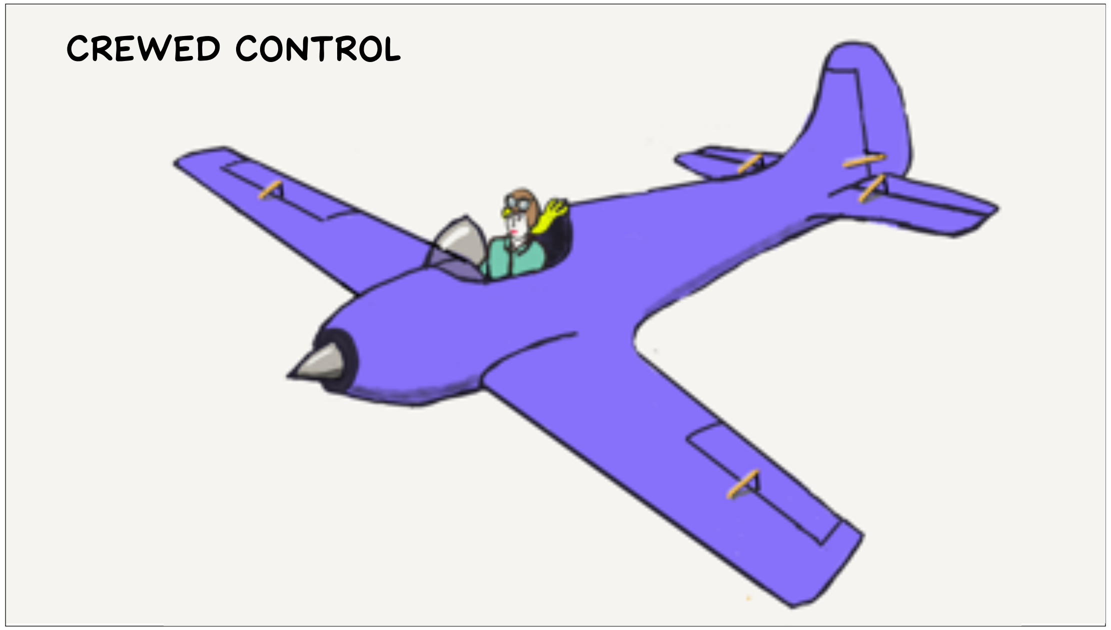
TALK CONTENT

- CONTROL/SENSOR BASICS
- AUTONOMY
- REQUIREMENTS
- HARDWARE
- COMMUNICATIONS
- SOFTWARE
- ELIXIR SWEET SPOT
- BRIEF RECORDED DEMO

I'll talk about the basics of how machines help to control vehicles and the kinds of sensors used; the nature of and requirements arising from autonomous vehicles; the hardware and communication methods commonly encountered; the programs you'll use and write, and what sort of programs Elixir in particular might be suitable for. We'll finish up with a simple demonstration that uses Elixir to wire up a physical remote control unit to a simulation. This is a lot to cover so let's start.



We’ll talk about autonomy later on – let’s start by talking about direct human control.

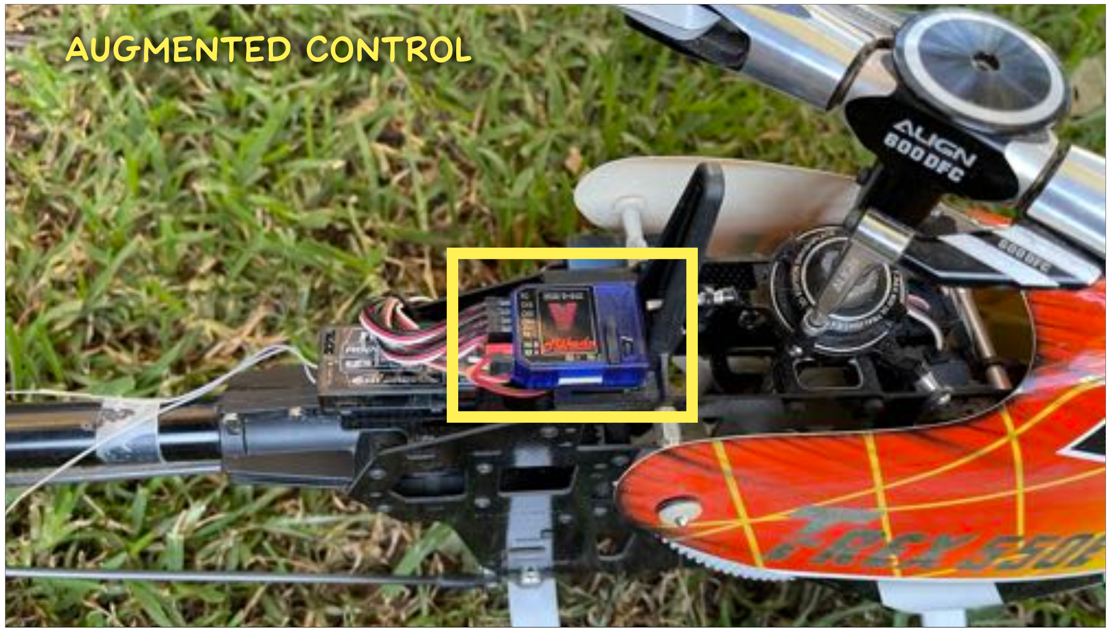


Aircraft started out as human carrying kites - Australian Lawrence Hargrave flew under his box kites at Stanwell Park south of Sydney, and the Wright Brothers flew their 1901 and 1902 gliders as kites before attempting free flight. An aircraft is really just a kite with the strings re-routed internally to a joystick and rudder pedals for the pilot to manipulate.

With remote control the pilot moves back to the ground and the strings are still there, it's just that a segment of their length is made out of radio waves.

In all of these cases we've seen direct, live control of the vehicle by a human using some sort of string-analogous technology. It's basically been a communications problem.

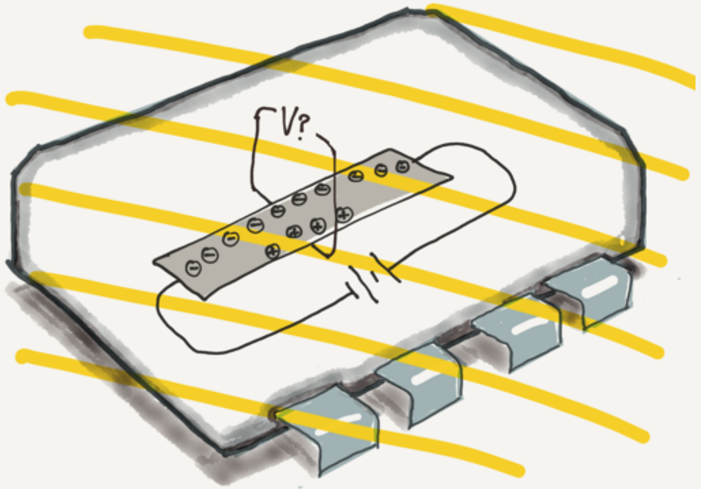
What happens when we want to use technology to start participating somehow in the control of the vehicle? For example, what if the pilot is learning to fly and needs help to avoid over controlling or getting into situations from which they cannot recover? Or what if the aircraft is so difficult to control that even a trained pilot needs some help to control it?



For example I have a large-ish RC helicopter. Helicopters are particularly difficult to fly because every control input causes one or two secondary reactions that you need to anticipate. I spent most of a year with a simulator unlearning my fixed–wing flying habits before attempting to fly this spinning carbon fibre vortex of death, and it still takes a lot of concentration. Like all model helicopters it has stability augmentation to reduce un-commanded changes in orientation - in this case it’s a processing unit highlighted by the yellow box sitting between the radio receiver and the servos (electric motors that control the rotor blade angle). Smaller helicopters would have the same thing integrated into the rest of the electronics on board.

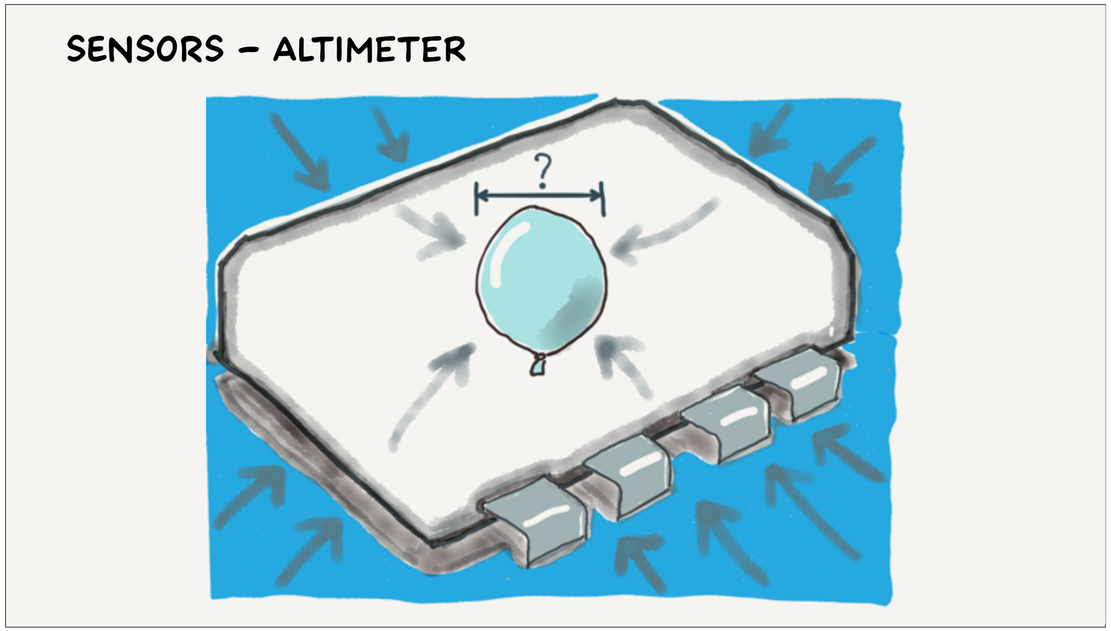
To make this work, the software on this box has to know what the pilot wants to happen; to sense what is actually happening, and then tweak the control outputs going to the servos and engine to try and reduce the error between the two. The first part (the pilot’s intentions) are coming from the radio receiver, so we need to talk about sensors and control outputs to complete the picture.

SENSORS – MAGNETOMETER



Magnetometer is a fancy word for compass, and it can be used to measure the orientation of the Earth's magnetic field in 3D, and from that the heading of the vehicle (an aside - heading is the direction you're pointing. Track is the direction you're travelling over the ground. They are generally not the same due to wind if you're in an aircraft, or currents and leeway in a boat).

I understand they use three "Hall Effect" sensors (I've only shown one here). These work by running a small electric current along the length of a conductor. The magnetic field causes the electrons to favour one side of the conductor over the other, and this bias can be measured by the voltage difference across the conductor, which is sort of cool.

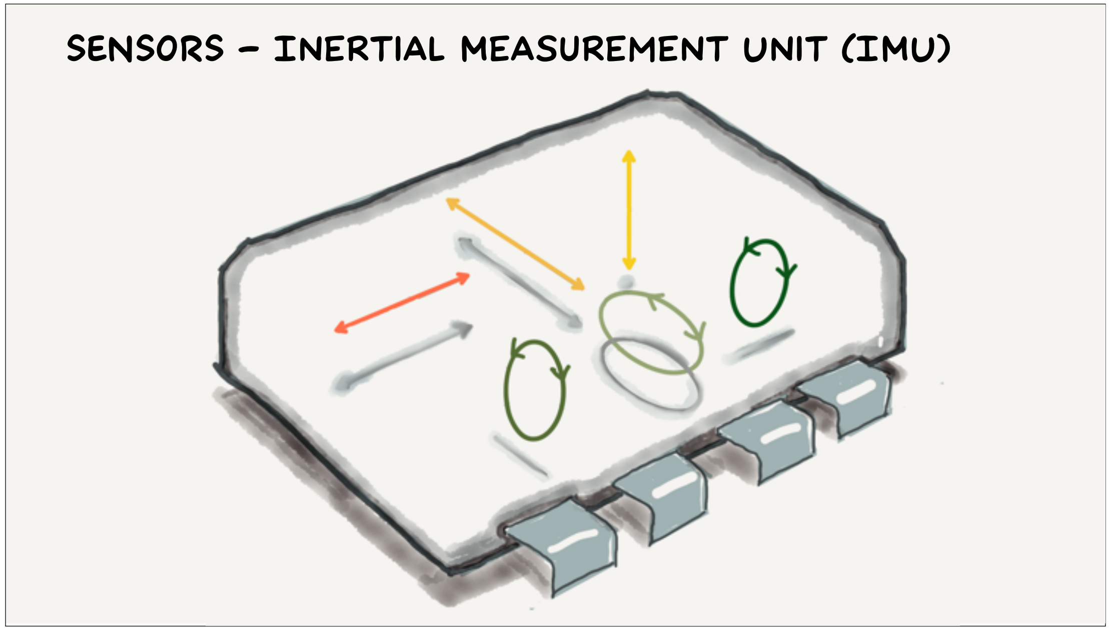


An altimeter or barometer measures the air pressure relative to some reference pressure - 1013 hectopascals is sea level in the standard atmosphere, and the pressure drops 1 hectopascal for every 30 feet you rise, until you reach space at just over 300,000 feet. It is actually a measurement of pressure difference, not height, and it's measured in feet even in countries using metric. The reason it is used is because it's easily available to all aircraft, who use it for collision avoidance.

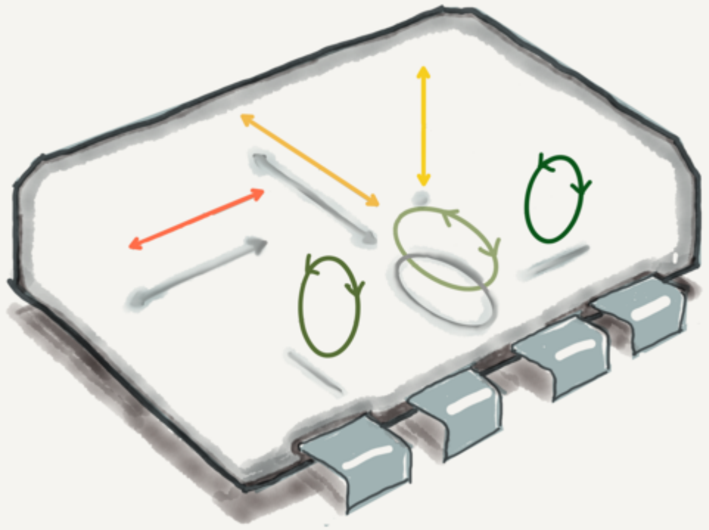
For this purpose your height above ground (as you might get from a GPS, or laser or radar pointed at the surface below you) is worse than useless, and changes with every hill and tree you fly over.

Anyway these chips have a little container of gas at a known pressure, and piezo crystals again determine how much the walls of the container are being bent by the atmosphere pressing in on it.

An airspeed sensor works similarly, except that it measures the pressure difference between the ambient air and the air being rammed into a forward pointing tube. This "dynamic" pressure is what makes the wings work, and is much more important to flight control than the speed over the ground, which is more relevant to navigation.

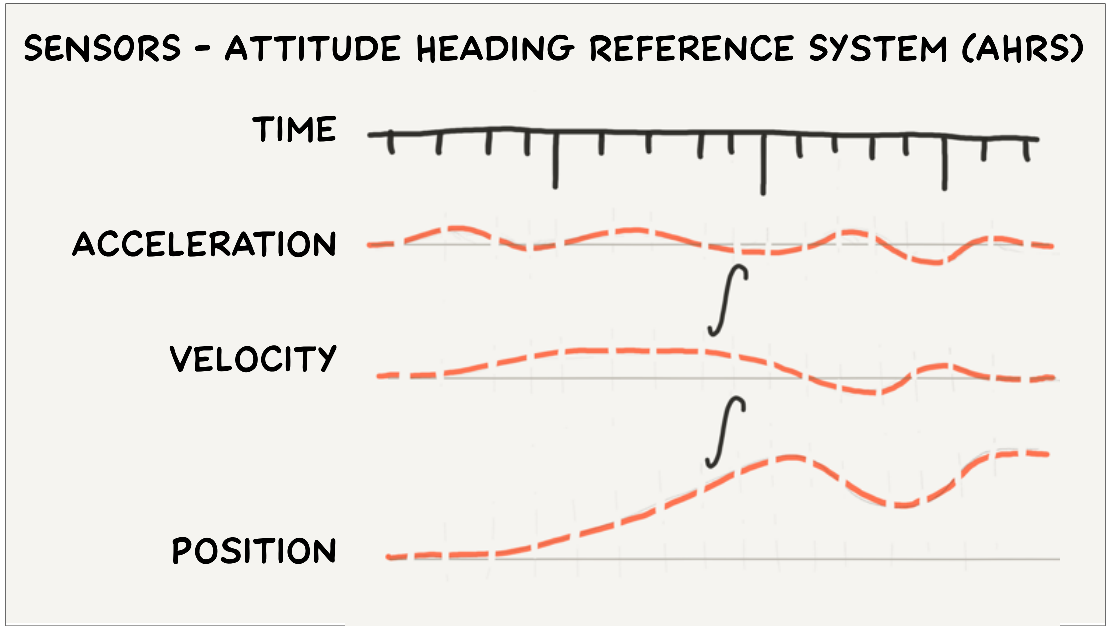


SENSORS – INERTIAL MEASUREMENT UNIT (IMU)



You have Inertial Measurement Units in your smartphone to measure acceleration and orientation. Most hobby UAVs use the same piezo electric IMUs found in consumer grade electronics - these measure voltages induced in crystals as they bend to give instantaneous measurements of acceleration and rotation rates in three axes.

There are higher accuracy alternatives such as laser ring gyros which are used in full scale aircraft and spacecraft but for our purposes the cheap ones are fine.

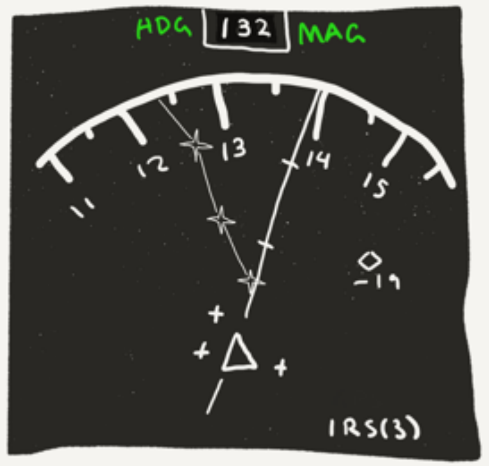


Instantaneous accelerations from the IMU aren't really useful in themselves. What you need to do is use a clock * to measure the acceleration * at regular intervals and keep a running total of all the acceleration measurements, which should be your current * velocity. If you do this again by summing up all your velocities you should get the * distance you've travelled.

Together with the heading from the magnetometer and the constant acceleration due to gravity if you know where you started you should be able to estimate your current position by dead reckoning (by the way dead is short for "deduced", and has nothing to do with pirates or revenge).

How well does this work? In the 90s I was on a 747 crossing the Pacific and turned out to be sitting next to an off duty pilot who invited me up to the cockpit to visit his colleagues while we were in a particularly remote bit of the flight southwest of Hawaii.

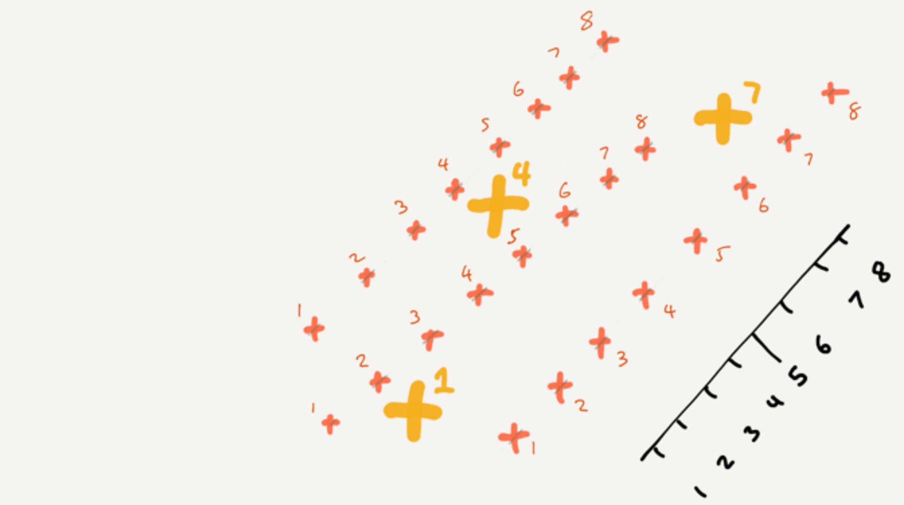
MORE THAN ONE SENSOR



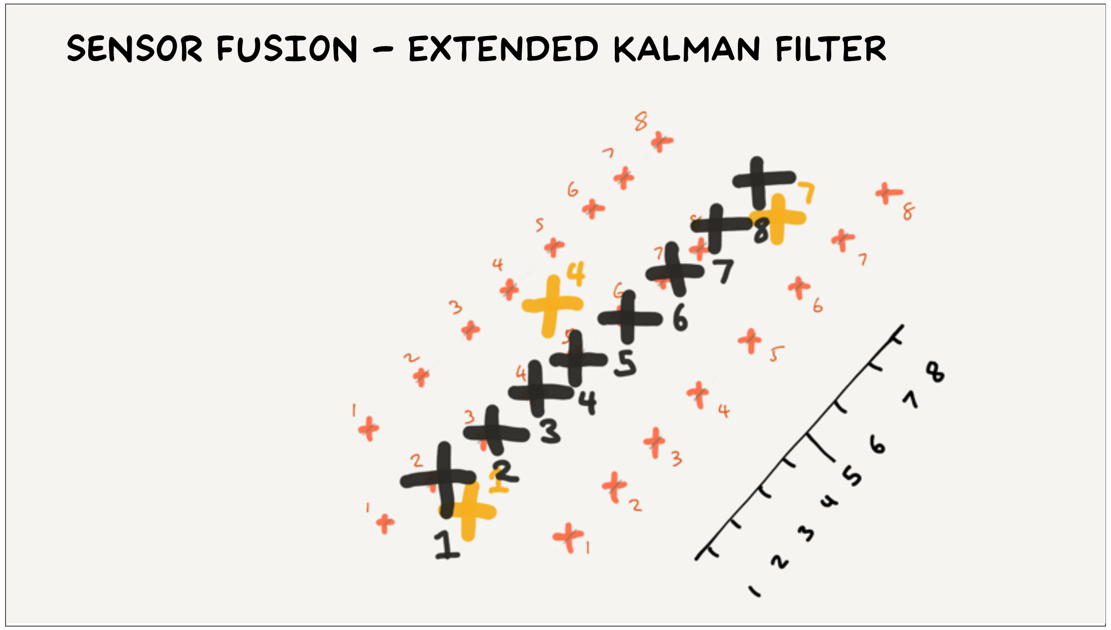
The 747 has three Inertial Reference Systems and their position was visible on the navigation display. At this point of the flight they were spread out across 10 nautical miles or so, which is pretty amazing considering they had been initialised in Sydney.

Consumer grade IMUs and AHRS units are a lot less accurate. I read one estimate saying that it would only take 20 or so seconds for your position to be out by more than 10 metres.

SENSOR FUSION



Luckily for us we live in the age of GPS so we can get position updates at a rate high enough to keep the AHRS errors in check. There are other sensors like LIDAR (distance measuring lasers) and motion flow (used by optical mice) that can help when you can't get a clear GPS signal. This leaves us with a problem though – how best to average out all the different estimates about our acceleration, velocity and position into an accurate estimate? This process is called "sensor fusion".



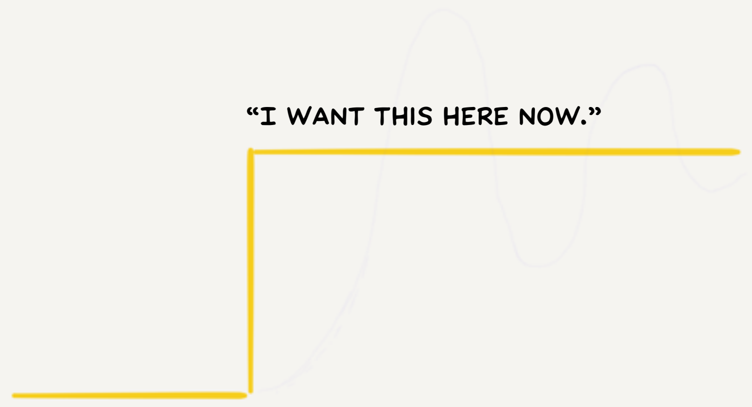
Wikipedia says that “the Extended Kalman Filter is arguably the de facto standard in navigation systems and GPS”. This sensor fusion algorithm is a child of the space age, with much of its development being undertaken by NASA because it’s exactly what you need to work out where your rocket is from lots of different sensors.

The EKF gives you a near optimal estimate of the current vehicle state that’s a mix of the new raw measurements with historical estimates.

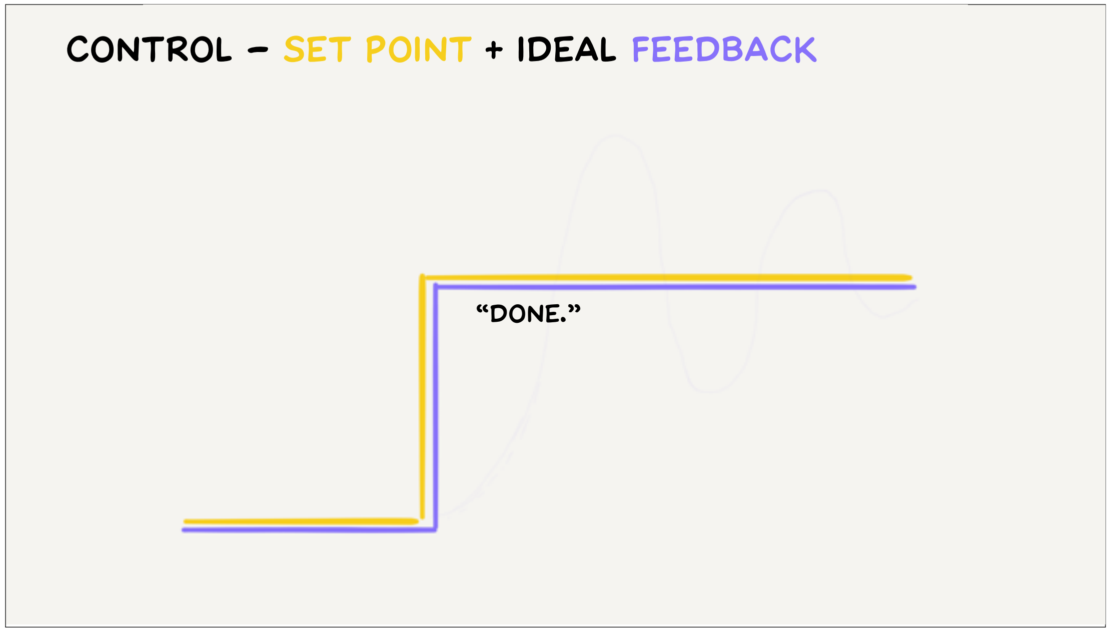
There are lots of Youtube videos and EKF libraries implemented in various languages. I couldn’t find one in Elixir, (there is one for vanilla Kalman Filters) but as I’ll explain later this doesn’t matter so much.

So we now know where the pilot wants to be, and approximately where we are

CONTROL – SET POINT

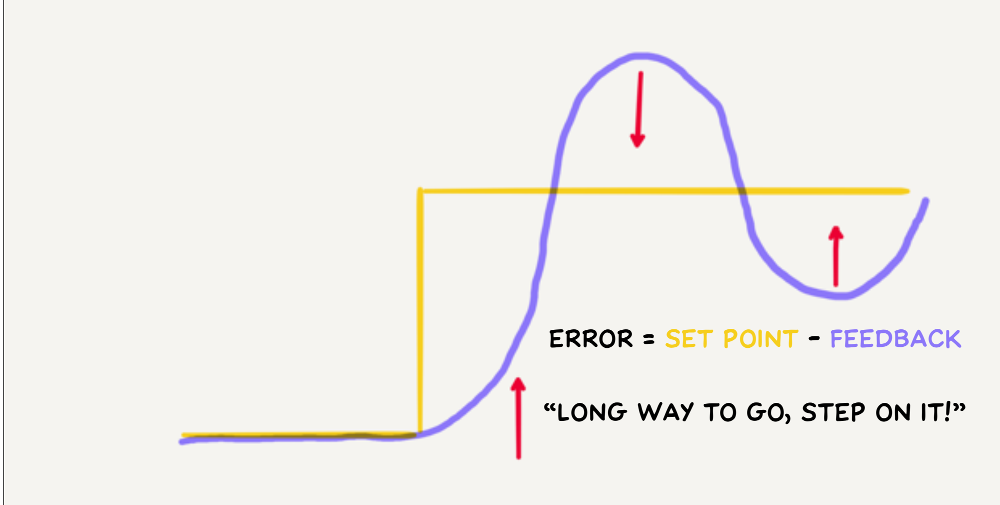


…so let's discuss how to close the gap between the two. Firstly our desired location is called the "set point".



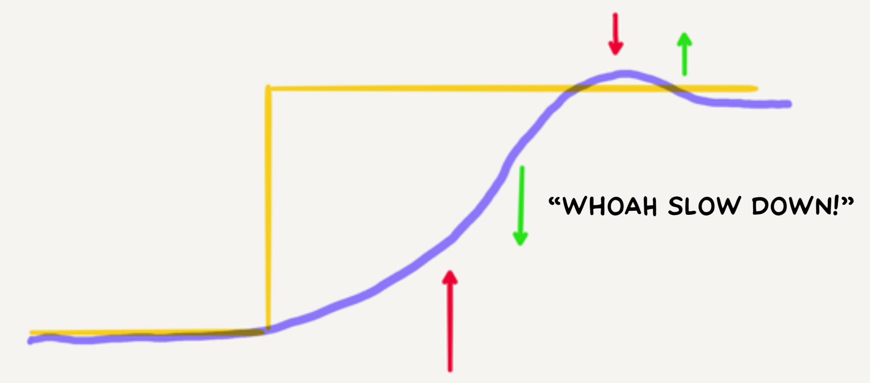
Under ideal circumstances with direct control and perfect sensor fusion the sensor feedback would match our set point. But neither of these things are true. In real life it's more like trying to steer a car or boat for the first time - you make a change, see what happens and adjust your input in a continuous loop. There are a few strategies you might follow to do this.

CONTROL – COMMAND PROPORTIONAL TO ERROR

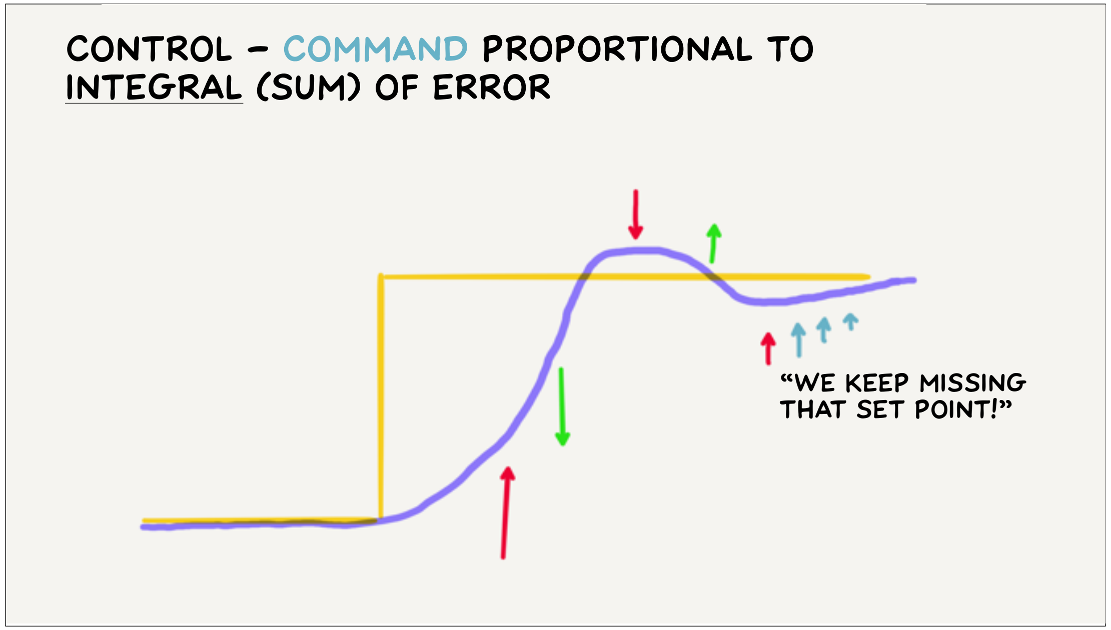


Firstly, the bigger the error between set point and feedback, the bigger the input you make. This generally leads to action-movie-style over control, particularly on a vehicle with lots of inertia like a bus or a boat. But if you dial in the proportion correctly it can work with a little bit of oscillation.

CONTROL – COMMAND PROPORTIONAL TO DERIVATIVE (RATE OF CHANGE) OF ERROR

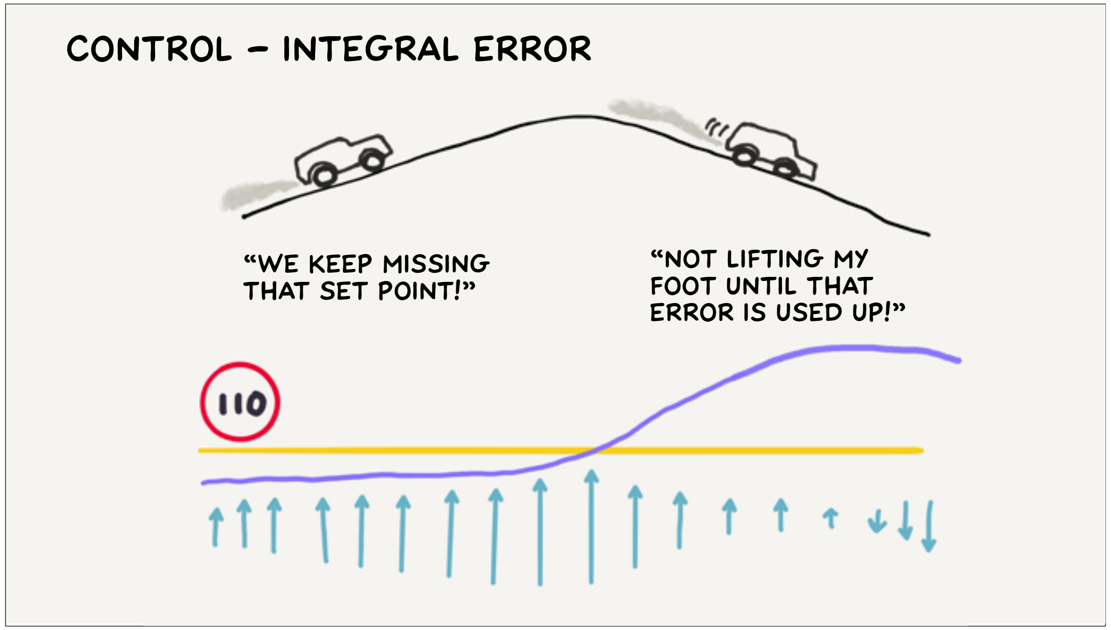


Secondly, the faster the error is changing, the more you put the brakes on to dampen the oscillations so that when you arrive at the set point you don't zoom past it. As this increases it takes longer to reach the set point, and if it's too strong it might smooth out your path to the point where important features (say your flight path hopping over rather than through a mountain) are left out.



CONTROL – COMMAND PROPORTIONAL TO INTEGRAL (SUM) OF ERROR

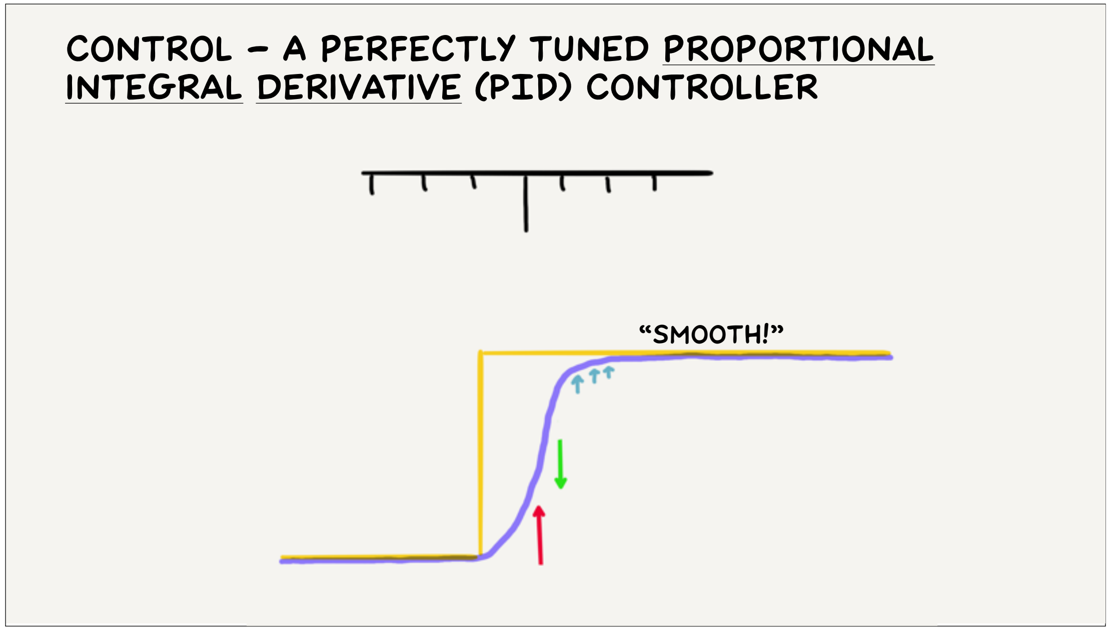
Finally, you can track over time whether you're consistently under or over-shooting your set point and introduce a pressure to correct it – an example might be steering a pram on a sloping footpath or controlling the height of a lift with a slightly stretched cable.


CONTROL – INTEGRAL ERROR
"WE KEEP MISSING THAT SET POINT!"
"NOT LIFTING MY FOOT UNTIL THAT ERROR IS USED UP!"
110

You have to be careful with this approach as error sums can build up such that when conditions change suddenly it takes a while to realise that the cause of the error is no longer there.

An example of this is the cruise control on our Forrester, which tends to overspeed as it crests hills with a long approach. At least that's what I plan to tell the Highway Patrol officer.

Another example is the alarming lurch my RC helicopter makes on take off from a non-level surface. The augmented stability unit has been trying to level the helicopter as the rotors spin up, but there's no thrust to make that happen and the total error keeps rising. As it lifts off it tries to cancel this error with extreme prejudice and overshoots a level attitude, at which point the proportional and derivative components take control, but for a fraction of a second the helicopter behaves like a dog about to roll over and scratch its back on the grass. To avoid this I either use a level surface or take off as quickly as possible to prevent too much error accumulating.

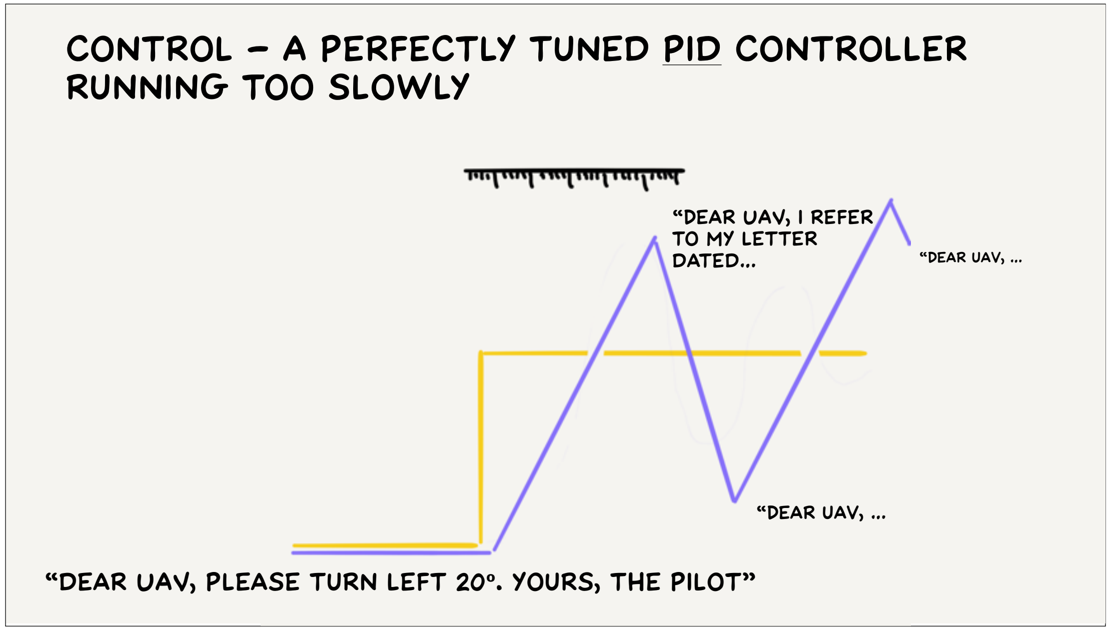


CONTROL – A PERFECTLY TUNED PROPORTIONAL INTEGRAL DERIVATIVE (PID) CONTROLLER

Together these three strategies are summed together to make a PID controller, used in autopilots, cruise control, air conditioners, lift motor controllers and pretty much anywhere software controls some physical process.

Tuning PIDs is a bit of a black art but there are processes to follow. Usually you start with P and add a bit of D to dampen things down. I is used sparingly for the reasons we just discussed.

The rate at which the sensor-fusion-PID-control loop runs is critical - generally the faster the better.



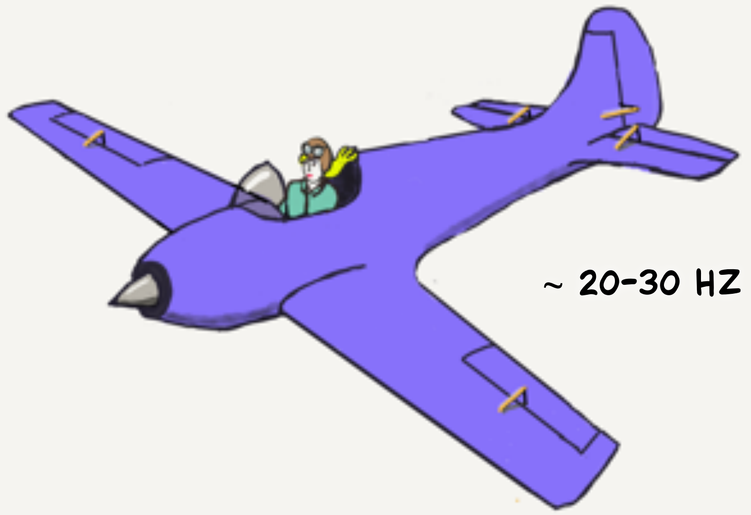
CONTROL – A PERFECTLY TUNED PID CONTROLLER RUNNING TOO SLOWLY

“DEAR UAV, I REFER TO MY LETTER DATED…
“DEAR UAV, …
“DEAR UAV, …
“DEAR UAV, PLEASE TURN LEFT 20°. YOURS, THE PILOT”

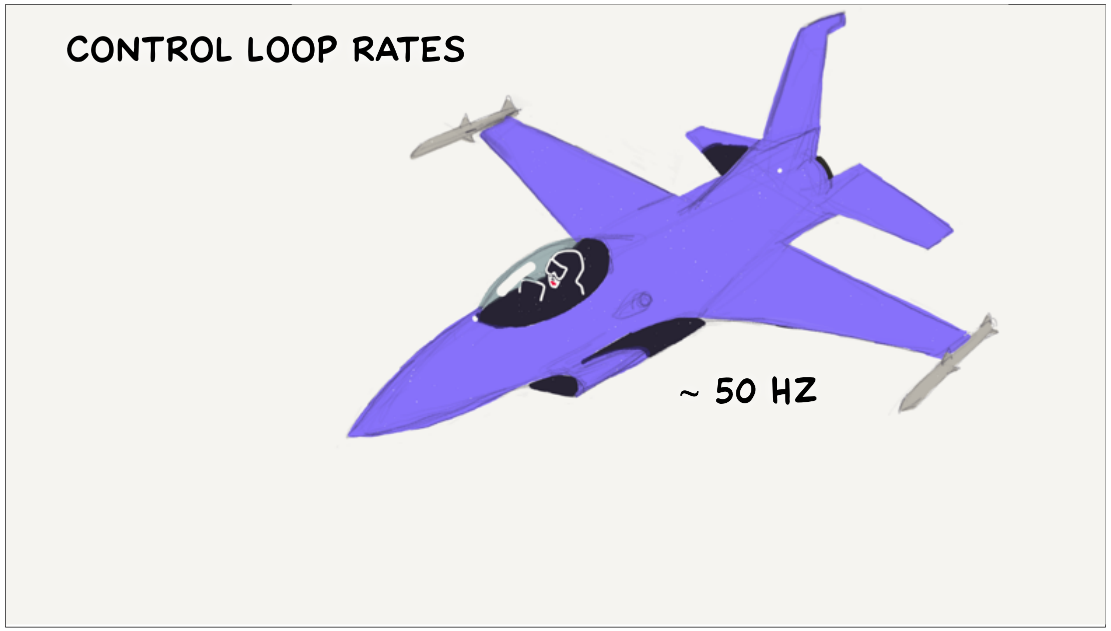
Too slow and chaos ensures - it’s like trying to fly via correspondence.

The required sensor-control loop speed differs depending on how stable or twitchy the thing is that you’re controlling.

CONTROL LOOP RATES



Most aircraft are designed to be naturally stable and have a fair bit of inertia and damping in all their axes of control. This makes them easy to fly - once trimmed in calm cruising conditions a pilot only needs to nudge the controls occasionally to maintain heading and height. As an aircraft becomes smaller and optimised for manoeuvrability (say a Red Bull aerobatic race plane) this is less and less true, however sensor-control loop rates of 20-30Hz should be sufficient.

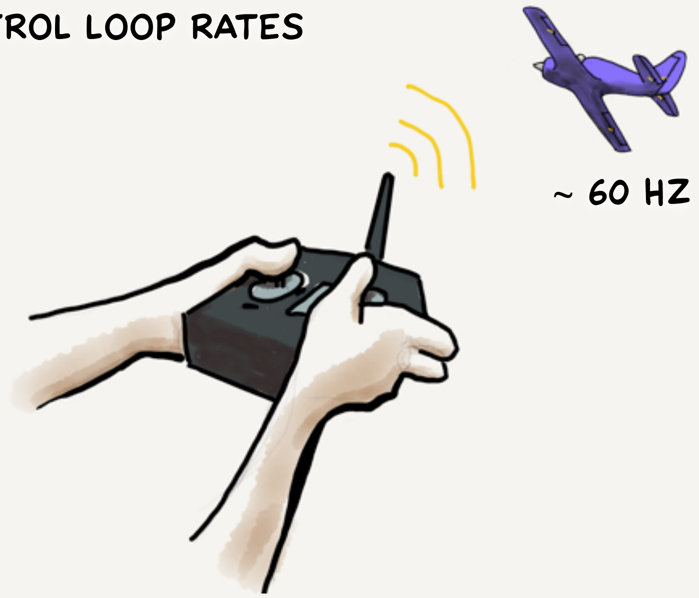


Where manoeuvrability becomes a matter of life and death, “fly-by-wire” controls allow military fighter aircraft to be designed from the outset to be unstable, and only controllable using something like the augmented stability unit on my RC helicopter.

Watching videos of aircraft carrier takeoffs you can see massive control inputs completely counter to what a pilot would normally command on takeoff, showing the PID hard at work keeping things together.

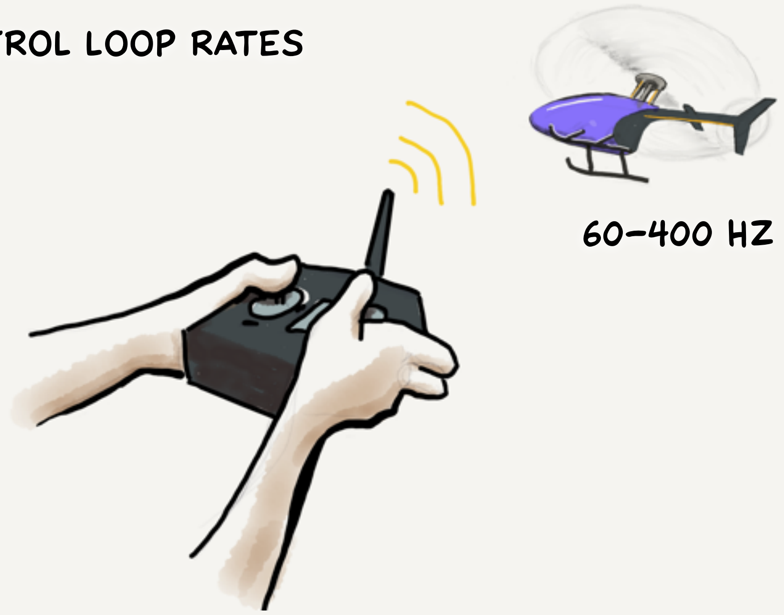
I recall reading that the General Dynamics F16 Falcon (designed in the 1970s) had a fly-by-wire system operating at 50Hz. Presumably more modern fighters are using rates at least as high and probably higher.

CONTROL LOOP RATES

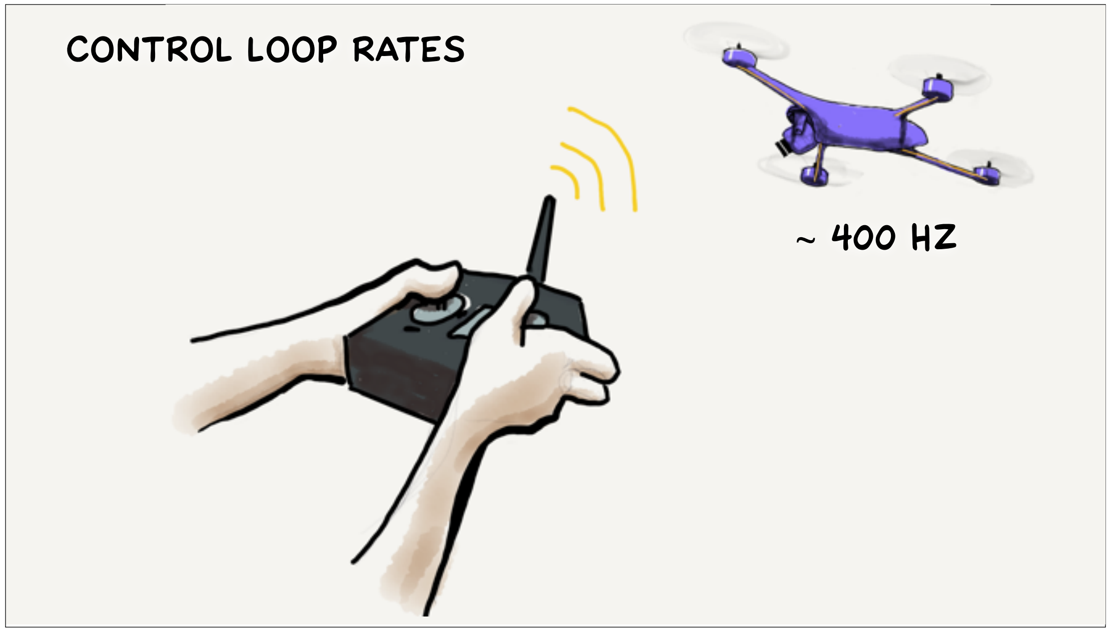


Model aircraft are less stable by virtue of their smaller size. When it ran on lower powered Arduino microcontroller hardware the “Ardupilot” autopilot software ran at 60Hz and it’s “Arduplane” control loop was sufficiently fast for fixed wing models.

CONTROL LOOP RATES



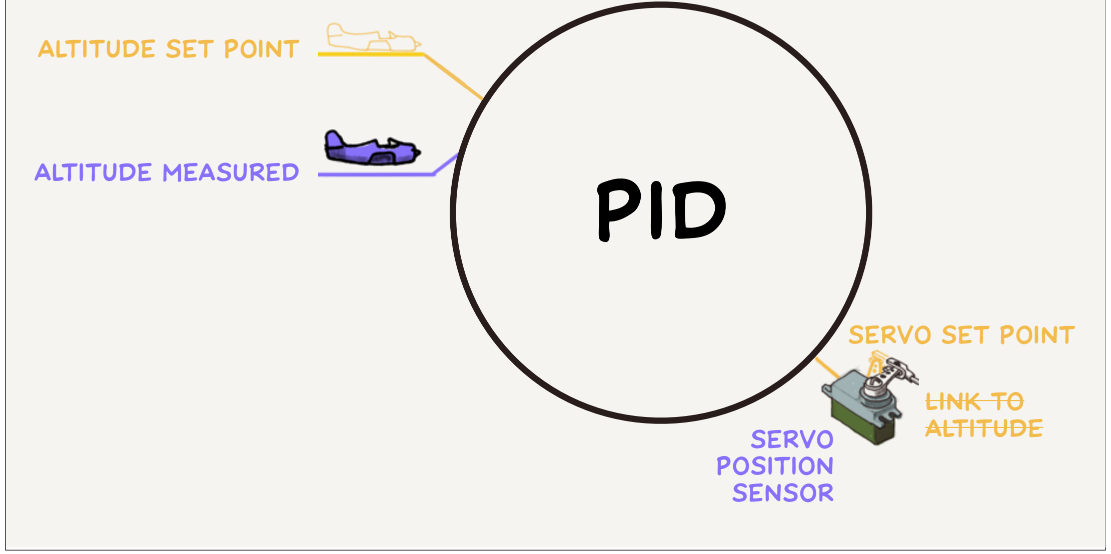
I never ran Arducopter on Arduino hardware but I understand 60Hz was right at the bottom end of what was possible for a “traditional helicopter” control.



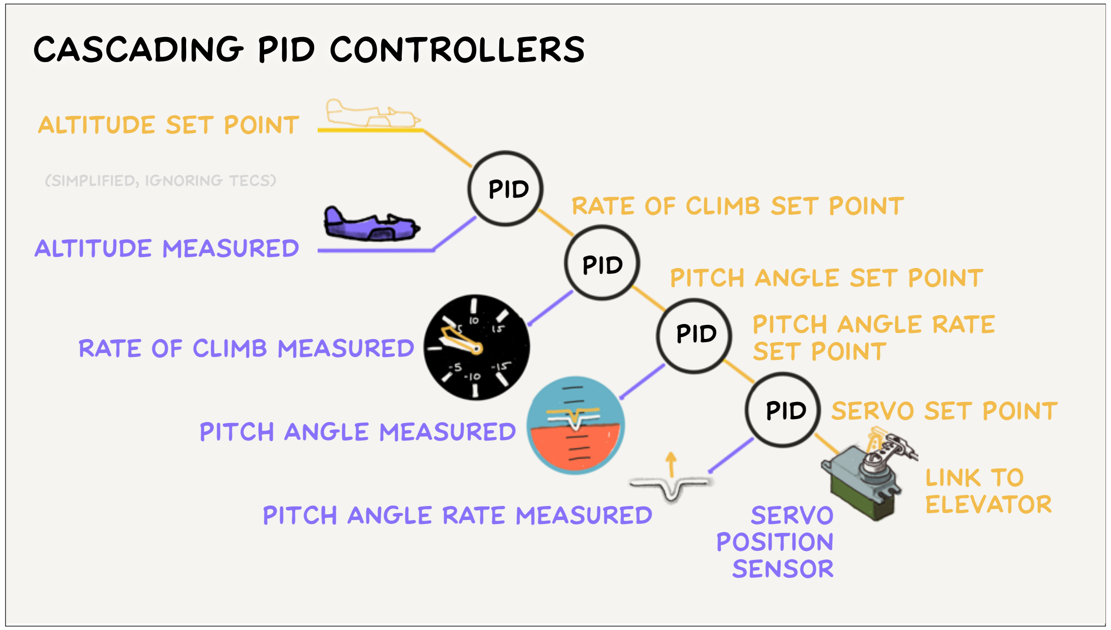
On faster hardware such as the Raspberry-Pi based Navio we ran on our 2016 “Quadplane” the control loop frequency was around 400Hz, and I understand that is typical now for most multi-rotor UAVs.

At 400Hz it should be clear that we are now talking about genuine real-time computing, with each loop taking 2.5 milliseconds.

INDIRECT CONTROL



Most PIDs aren't directly controlling the thing being measured. For example, if we want our UAV to fly at a particular altitude, it's not like the whole UAV is on the end of a string tied directly to a servo arm.

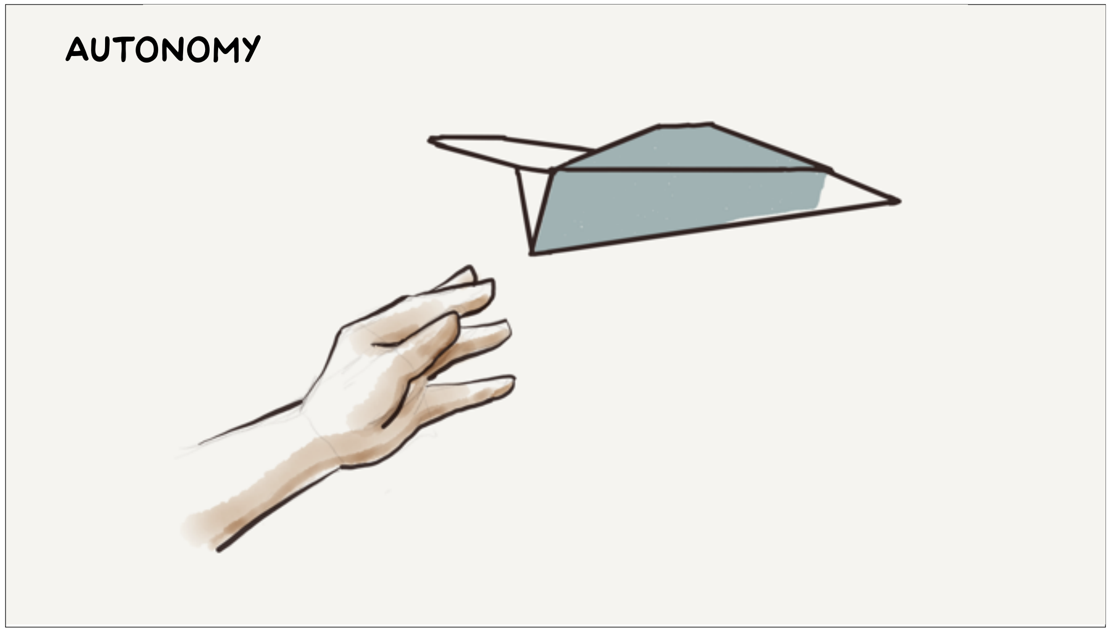


Instead the servo is connected to the elevator, a flying control surface on the horizontal stabiliser at the back of the aircraft that determines how quickly the aircraft is pitching nose up or nose down.

This means that the first PID tries to fix the altitude using the rate of climb, the second PID tries to fix rate of climb with a pitch angle. The third PID tries to fix pitch angle with the pitch angle change rate and the final PID tries to fix the pitch angle change rates using the elevator position.

The servo itself has a sensor that measures its current arm position and engages the motor to wind one way or the other to match the set point, making it a little hardware sensor-PID loop in its own right.

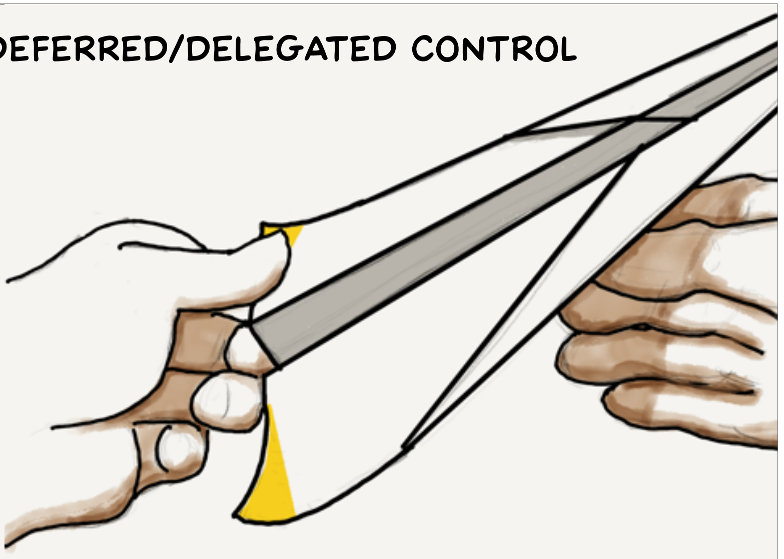
That's it for our basic introduction to sensing and controlling vehicles. While you probably will not have to program this stuff yourself, it explains a cool bit of technology that lets tiny camera drones hover like they're nailed to the sky and SpaceX boosters nail landings from space. And it highlights the safety critical nature of tuning PIDs (which you will be doing) and giving them the real-time computing resources they need to work. (~15 MINUTES AT FAST PACE)



AUTONOMY

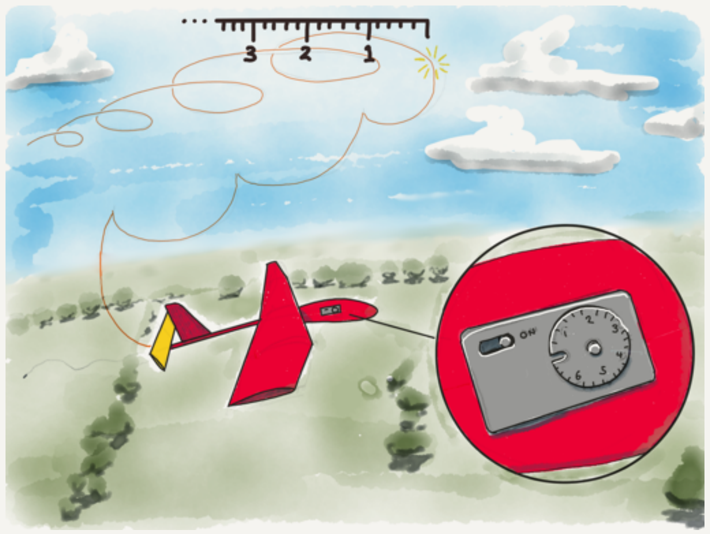
Autonomy is about what happens when a human has left the live control loop.

AUTONOMY – DEFERRED/DELEGATED CONTROL

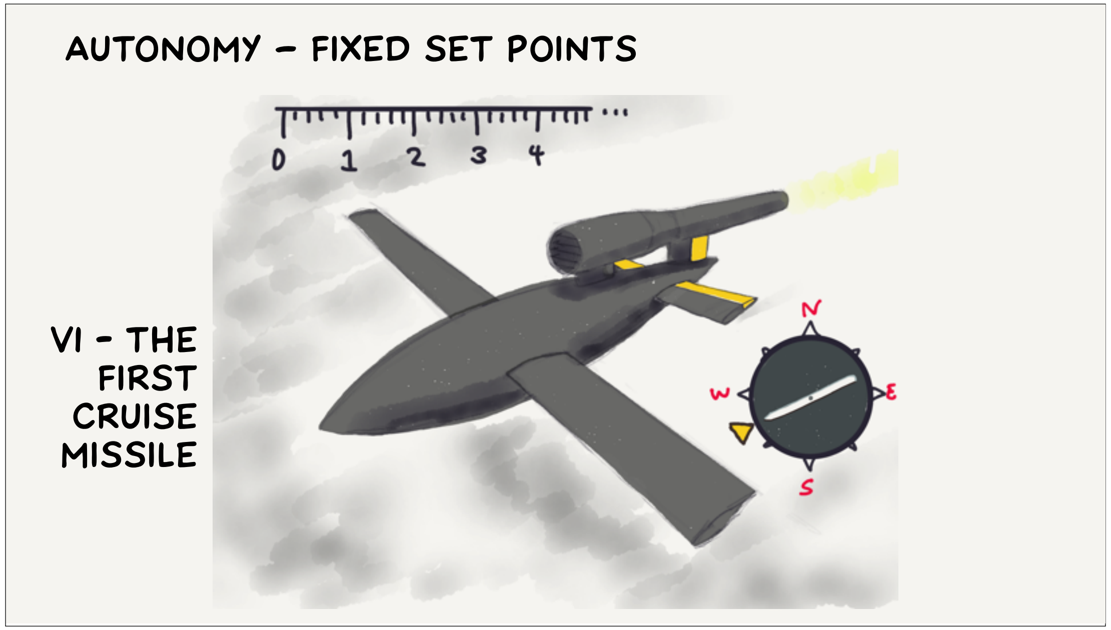


There is still control, but it's set up before the flight. There's a blurry line between the design process and issuing autonomous commands, but it becomes clearer…

AUTONOMY – SENSORS TRIGGER ACTIONS

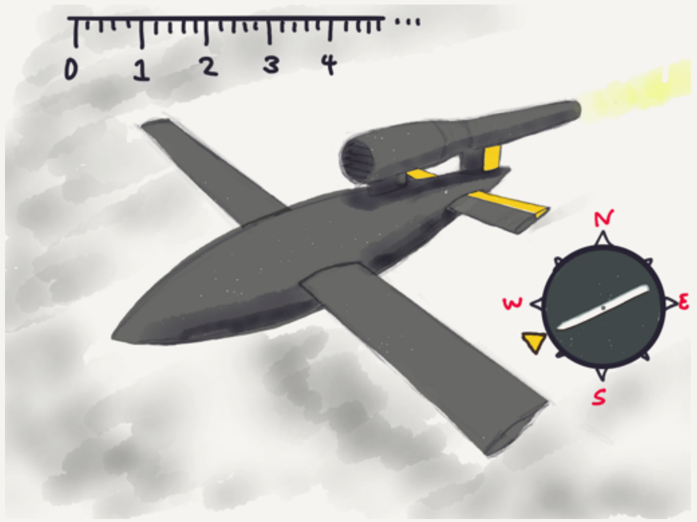


as time based triggers become involved, for example the clockwork timers they use on free flight glider models to cause them to return to earth after a set period of time.

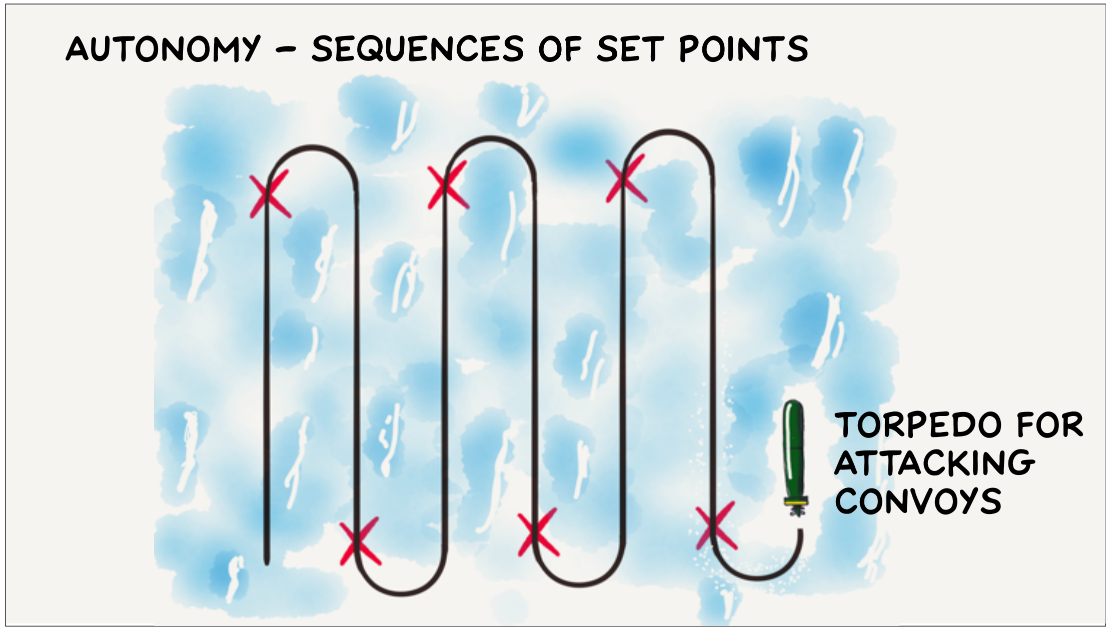


AUTONOMY – FIXED SET POINTS

VI - THE FIRST CRUISE MISSILE

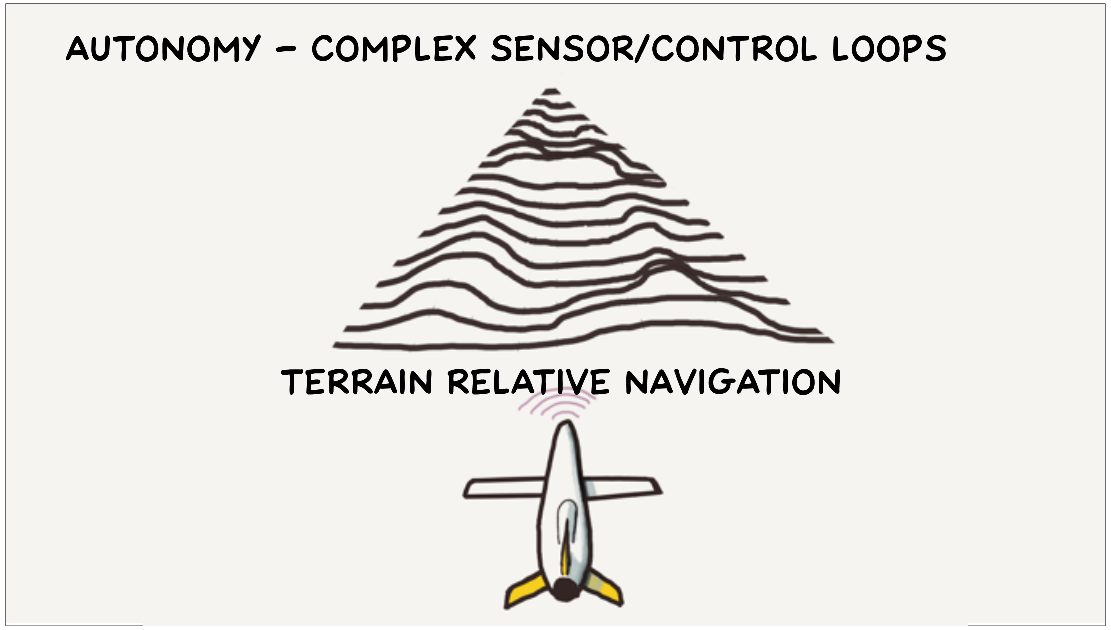


The same sorts of sensor control loops we've talked about can be used with a set point fixed before launch…



AUTONOMY – SEQUENCES OF SET POINTS

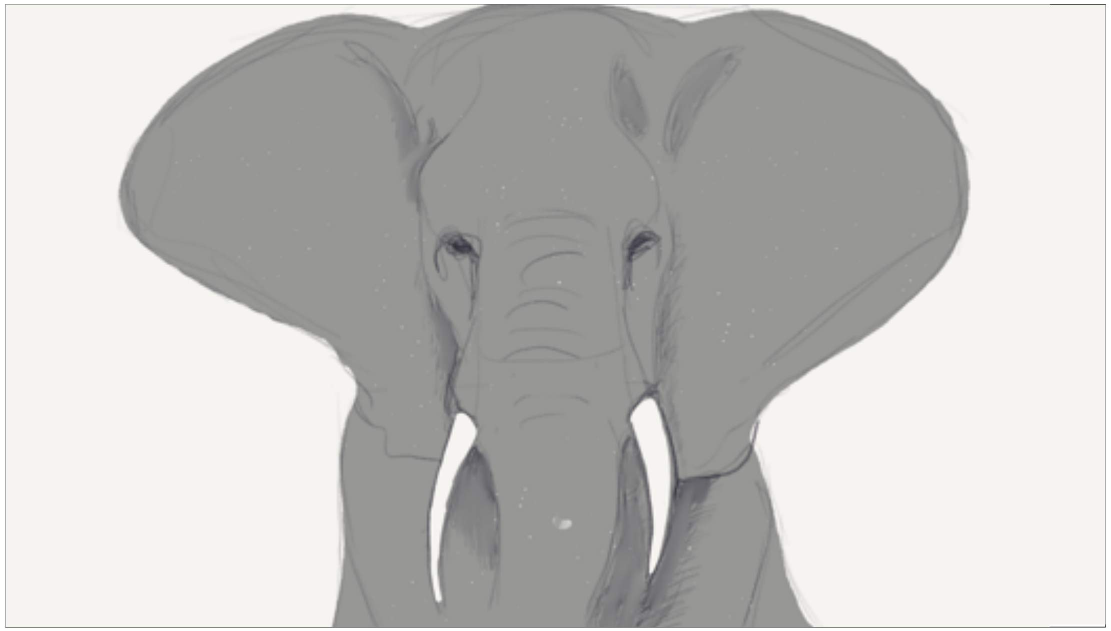
Or the set points can vary according to a timer, or advance as the next navigation waypoint on a flight plan established before takeoff is reached.



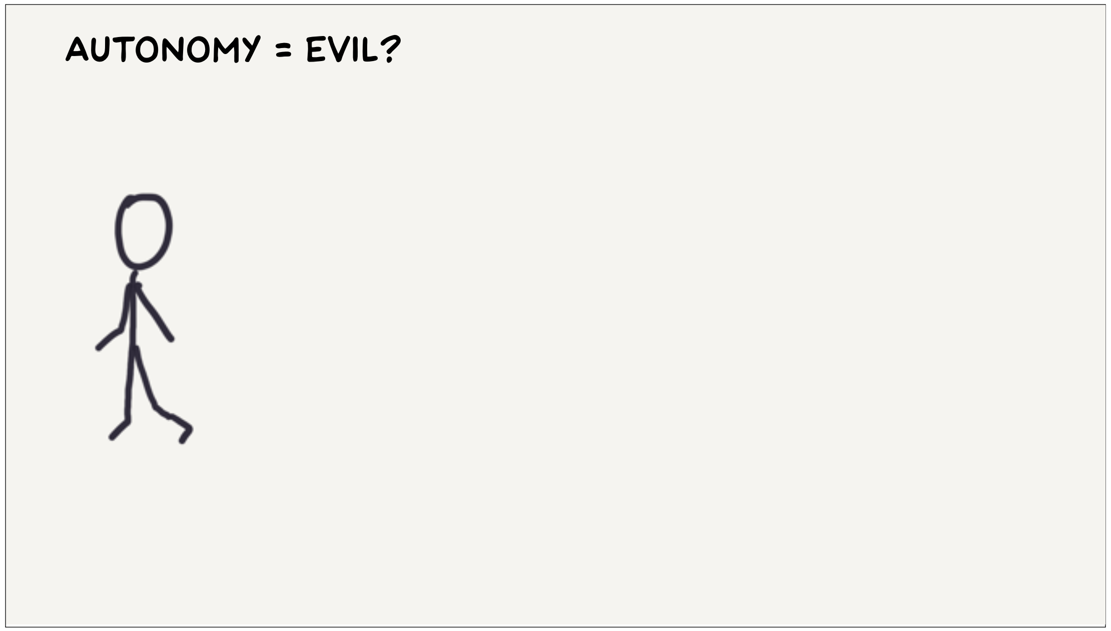
AUTONOMY – COMPLEX SENSOR/CONTROL LOOPS

TERRAIN RELATIVE NAVIGATION

Set points can be updated continuously from a recording (say robot dance moves) or by real time sensors, such as the ground scanning radars used by modern cruise missiles to navigate without relying on vulnerable satellite navigation networks.



Since we're talking about autonomy and I seem to have struggled over the last few slides to find some more pleasant use cases we may as well address the elephant in the room…



AUTONOMY = EVIL?

… is autonomy evil? A landmine is an example of an autonomous weapon with a single sensor and a single response. It will explode and kill or maim anyone or anything that touches it. It is widely considered to be one of the most awful and insidious kinds of weapon ever created.

AUTONOMY = EVIL?

…what if we used a camera and a computer vision classifier sensor so that the mine was only detonated by a person wearing a military uniform - is that more - or less, evil?

AUTONOMY = EVIL?

What if we restricted the criteria even further so that they had to be armed?

AUTONOMY = EVIL?

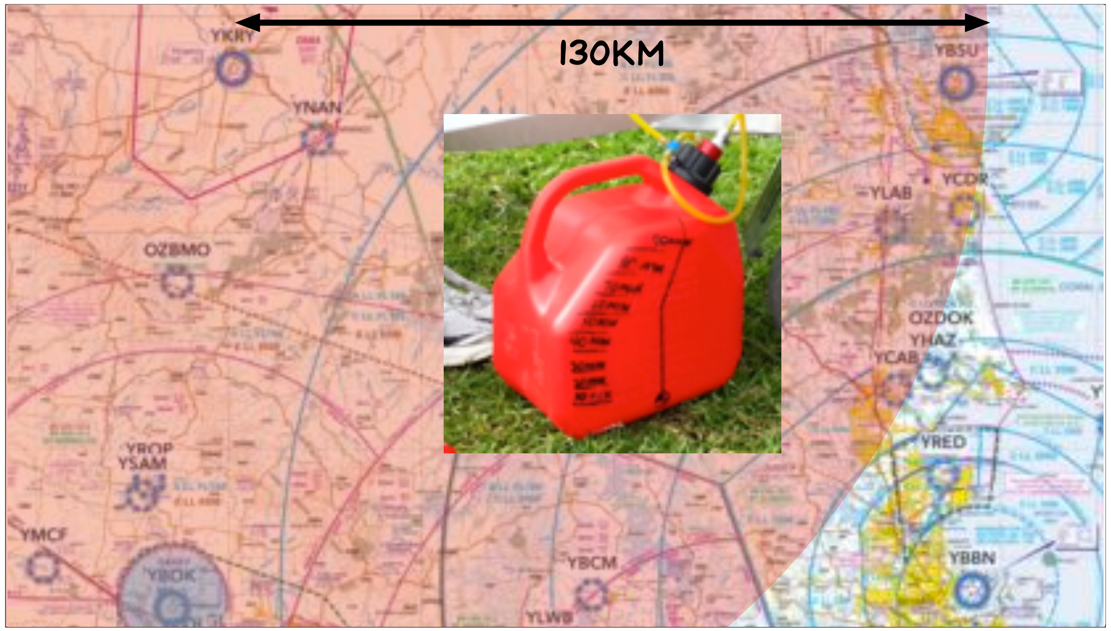
SARAH CONNOR

Or used face recognition to target a specific list of people? I'm absolutely not making the "guns don't kill people" argument. I'm more cautioning against falling into Terminator tropes and allowing a little more room for nuance the next time you're in the comments section of a Boston Dynamics robot video. Of course in civilian life…

AUTONOMY – FUNDAMENTAL REQUIREMENT

…we don't want to kill, maim or even mildly inconvenience ANYONE. This is the fundamental requirement of UAV autonomy, and it's what the vast bulk of autonomous behaviour in UAVs is designed to mitigate. Taking photos and delivering packages are a distant nice-to-have second item on our autonomy priority list.

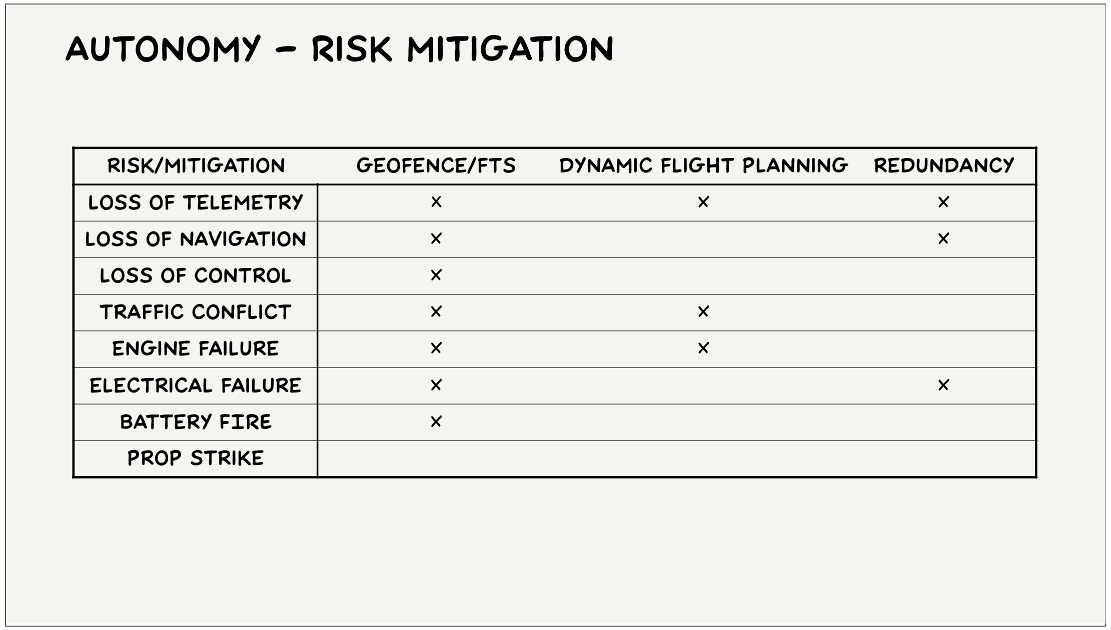
UAVs are not toys and the risk is real. For example…



…in it’s 2014 Outback Challenge configuration our UAV weighed 26kg and carried * 5 litres of petrol, which surprisingly (and I checked twice) has the same energy content as 170kg of TNT.

From the competition location at Kingaroy it had the range to reach the Sunshine Coast to the east or Toowoomba to the south - an area containing 100s of 1000s of people and vast tracts of inflammable bushland.

Model helicopters have killed people. An unpowered 2kg drone can badly injure someone just falling on them.

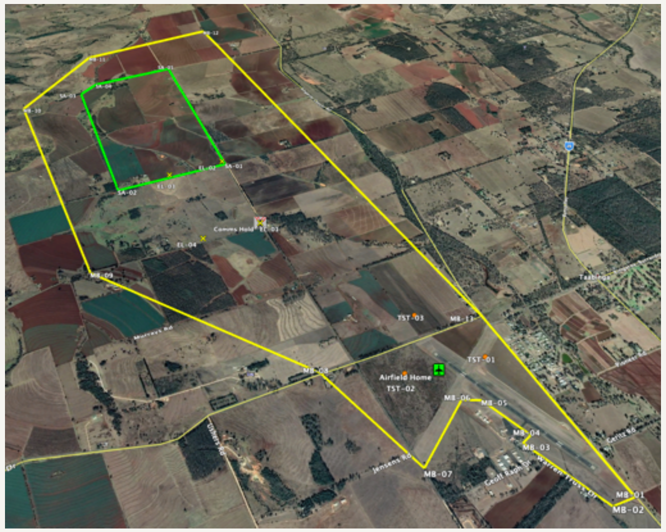


AUTONOMY – RISK MITIGATION

RISK/MITIGATION	GEOFENCE/FTS	DYNAMIC FLIGHT PLANNING	REDUNDANCY
LOSS OF TELEMETRY	×	×	×
LOSS OF NAVIGATION	×		×
LOSS OF CONTROL	×		
TRAFFIC CONFLICT	×	×	
ENGINE FAILURE	×	×	
ELECTRICAL FAILURE	×		×
BATTERY FIRE	×		
PROP STRIKE			

What are some specific ways in which our UAV mission can go pear shaped?
We could loose our ground station or communication link with the aircraft.
We could loose a GPS signal or altimeter and become uncertain of our position.
We could have a blocked airspeed pitot tube, an IMU failure or badly tuned PIDs and loose control of the aircraft.
We could find ourselves in conflict with other air traffic, loose an engine or electrical bus, or one of our lithium polymer batteries could short and catch fire (as they are prone to do).
Some risks such as prop strikes are addressable with team procedures and checklists, but after takeoff the onboard hardware and software are only way to affect the outcome of the mission.

GEOFENCE/FLIGHT TERMINATION SYSTEM



The best way to protect people from a UAV is to fly it somewhere where they are not - just one reason urban delivery drones are a really dubious idea. A geofence (like the one shown from the 2014 challenge) is a boundary set before launch containing the entire flight plan .

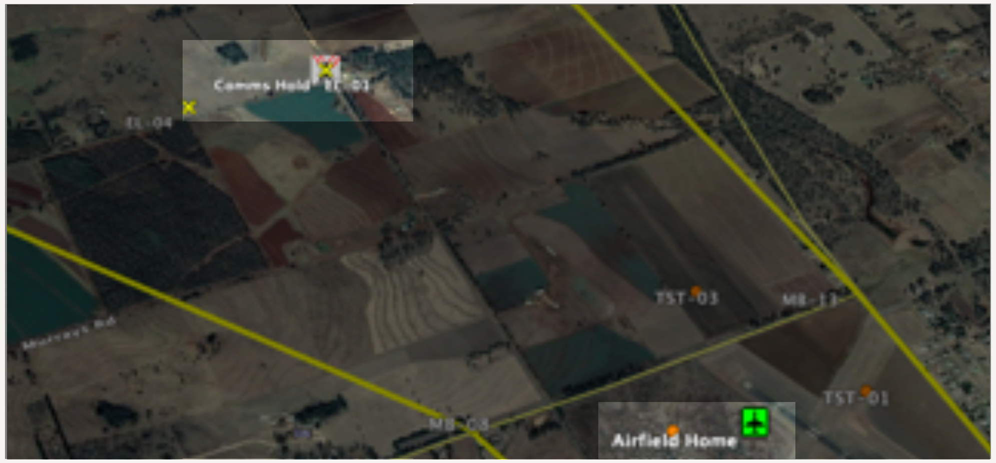
Geofences are separated into “hard” and “soft” varieties. A soft geofence might trigger a return to base or turn back towards the main volume of the geofenced area, assuming that there’s perhaps a strong wind or error in the flight plan, and that the operators will issue new instructions. A hard geofence enclosing the soft geofence will trigger the flight termination system, based on the assumption that whatever was supposed to happen at the soft geofence hasn’t worked, and the risk to people and property inside the geofence of a UAV planting itself in the ground is lower than the risk outside.

For the competition notices are published to global pilot briefing systems warning of UAV activity, residents have given permission for UAV operations over their properties and have been briefed on the risks. The geofence boundary was just behind the team set-up tent at the airfield and part of the qualification check was for each team to carry their aircraft over the geofence so that the competition organisers could confirm the triggering of the FTS.

GEOFENCE/FLIGHT TERMINATION SYSTEM

Flight termination can be triggered for other reasons than a hard geofence breach, for example a failure of the GPS signal, which will cause the fused position estimate to be unusable within 30 or so seconds. Termination involves sending set points directly to the servos to cut the engines on a helicopter or multi-rotor, or initiate a spin on a fixed wing aircraft. A spin is better than a dive because while the flight path is still vertical, the speed is limited and less likely to cause an impact fire or damage property. That said, our own UAV demonstrated to us during an unscheduled in-flight reboot of one of its computers that even without FTS inputs it was going to make a crater pretty much directly underneath where the computer failed. I’m sorry I can’t find the video, it involved a lot of screaming and a very very late save.

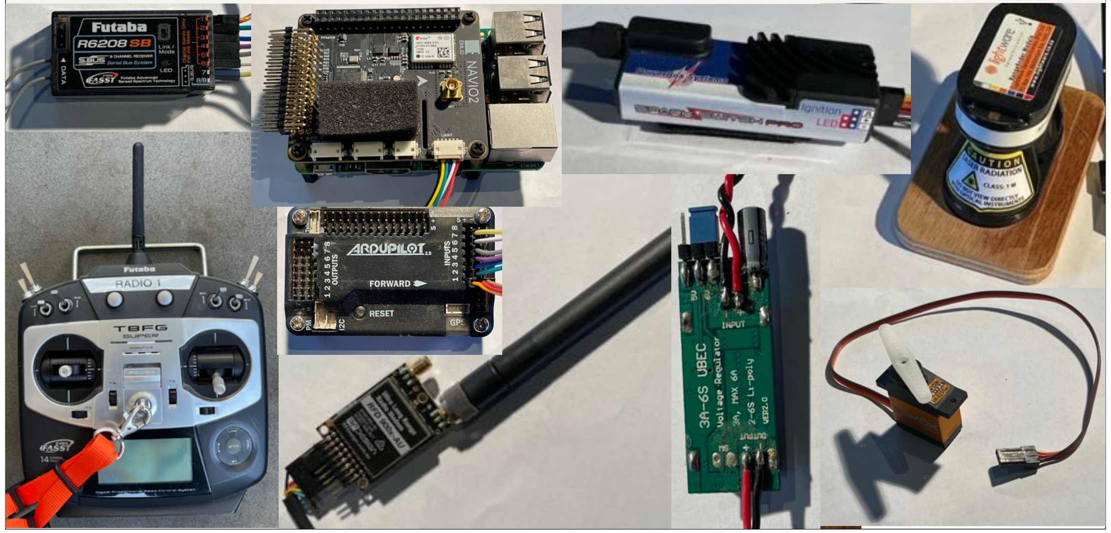
DYNAMIC FLIGHT PLANNING



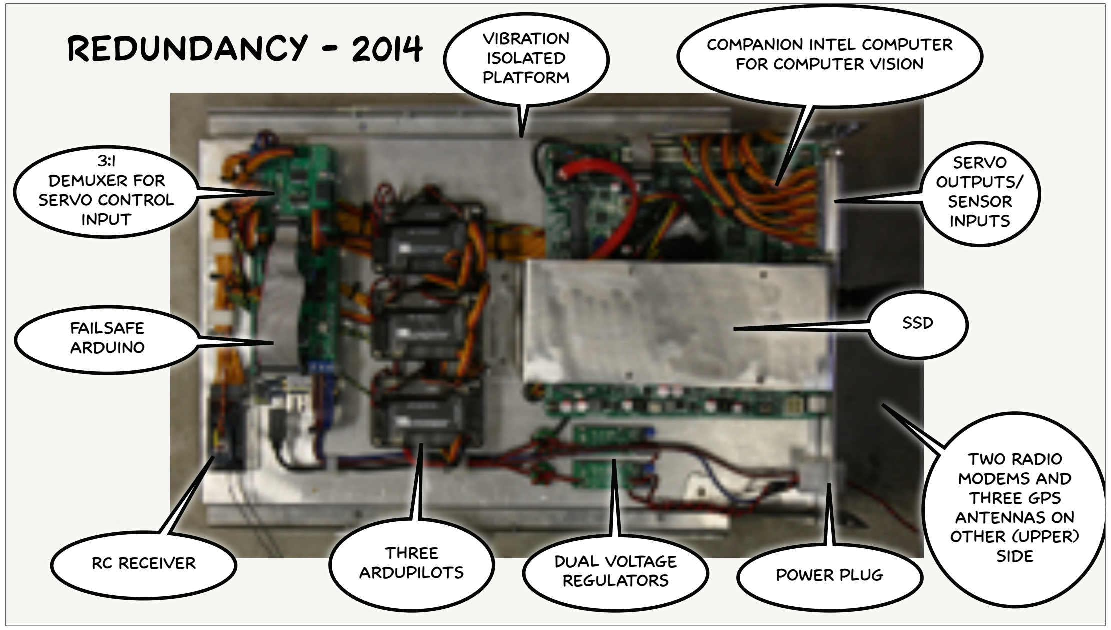
In the case of a communications failure the UAV is still flyable, but the mission priority shifts to re-establishing communications. This means reducing the distance to the ground station - the most simple option is to skip to a point in the flight plan where the aircraft returns to launch and orbit overhead, or where the distance is greater, at an interim hold point between the mission area and the launch. Eventually FTS or a landing (the latter much easier for helicopters than fixed wing to perform autonomously) can be triggered after a timeout. The advantage of a comms hold point is then that the FTS isn't triggered over the operator's heads.

In the most recent competitions there has been an "extension challenge" to autonomously avoid simulated moving flocks of birds, bad weather and aircraft transiting the mission airspace, while continuing the mission and not breaching the geofence. This involves forward planning, projection of flight paths and constant battles of priority with all sorts of nasty edge cases to test. It's hard enough with these challenges, let alone cars, children and dogs: another reason urban delivery drones are a really really dubious idea. And yet this sort of autonomy is a really interesting problem, and right at the cutting edge of hobbyist, academic and commercial research.

REDUNDANCY - QUICK INTRO TO HARDWARE

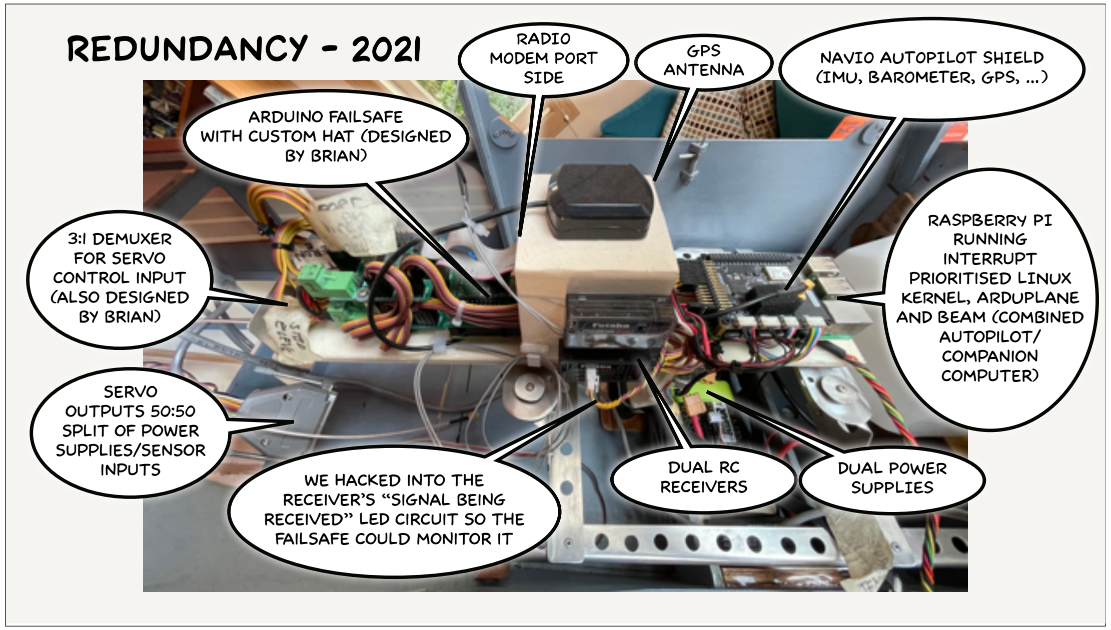


Before I show you more avionics pictures here’s a quick identification guide:
Model aircraft radio control * receivers and * transmitters, which work on the same 2.4GHz range as WiFi.
* A radio modem with an omnidirectional antenna. Depending on distance and the gain of the antennas a pair of radio modems can provide a serial link from the USB port of a laptop running ground station software to the UAV. This particular radio modem is Australian designed and uses the 920MHz class license frequency range.
* An original Arduino microcontroller based Ardupilot
* A more recent Raspberry PI based Ardupilot with a NAVIO 2 “shield” or “hat” module plugged in containing IMUs, altimeters, and connections to modems and other sensors
* An engine remote ignition switch and sensor package that monitors RPM and the separate ignition battery voltage while isolating ignition electrical noise from the rest of the avionics (that stumped us for a while: the closer you get to hardware from software the more unfair the bugs get)
* A LIDAR for accurate height measurement above ground, very necessary for landing fixed wing UAVs
* a voltage regulator to provide the correct amount of power from the batteries to your hardware, and
* a servo used to move flight control surfaces


REDUNDANCY - 2014
VIBRATION ISOLATED PLATFORM
COMPANION INTEL COMPUTER FOR COMPUTER VISION
3:1 DEMUXER FOR SERVO CONTROL INPUT
SERVO OUTPUTS/ SENSOR INPUTS
FAILSAFE ARDUINO
SSD
TWO RADIO MODEMS AND THREE GPS ANTENNAS ON OTHER (UPPER) SIDE
RC RECEIVER
THREE ARDUPILOTS
DUAL VOLTAGE REGULATORS
POWER PLUG

This is the avionics platform for our 2014 entry, with five separate computers: three autopilots, a companion computer for computer vision tasks and a failsafe arduino. The failsafe had an independent power source and ran a state machine that monitored the status of the autopilots, communications and navigation, and controlled which of autopilots or the RC receiver got to control the servos through the demuxer circuit. If the answer was none of them, it would send the flight termination outputs to the servos itself.

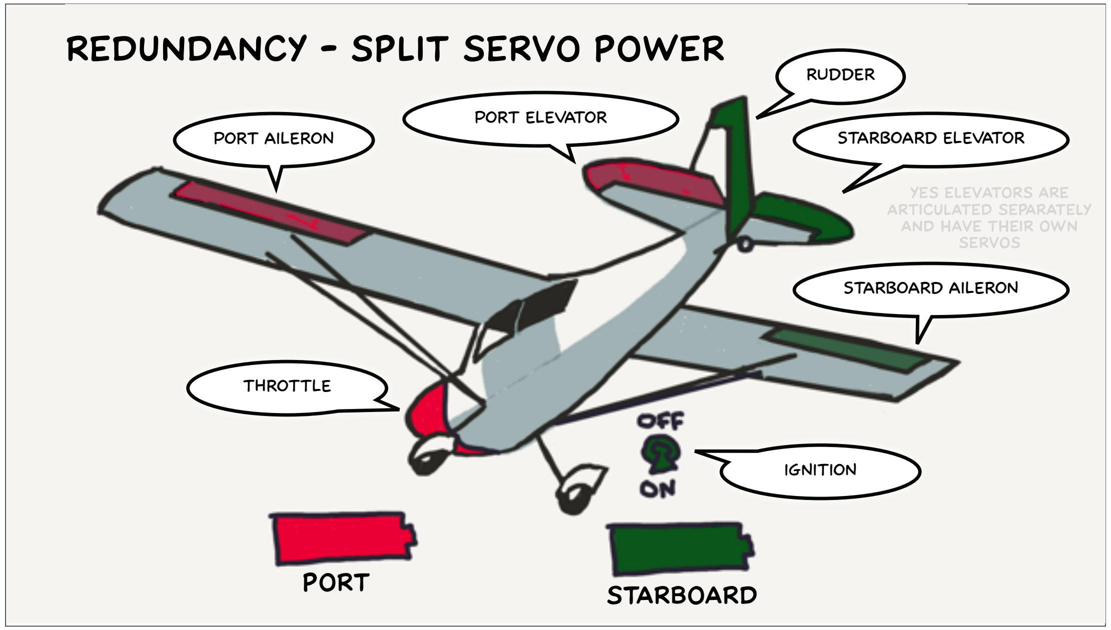
This may go some way to explaining why our two person team ran out of time to make the competition deadline (blowing up my car's electrical system trying to power our ground station trailer and problems with the UAV's engine ignition unit didn't help either).


REDUNDANCY - 2021
RADIO MODEM PORT SIDE
GPS ANTENNA
NAVIO AUTOPILOT SHIELD (IMU, BAROMETER, GPS, ...)
ARDUINO FAILSAFE WITH CUSTOM HAT (DESIGNED BY BRIAN)
3:1 DEMUXER FOR SERVO CONTROL INPUT (ALSO DESIGNED BY BRIAN)
RASPBERRY PI RUNNING INTERRUPT PRIORITISED LINUX KERNEL, ARDUPLANE AND BEAM (COMBINED AUTOPILOT/ COMPANION COMPUTER)
SERVO OUTPUTS 50:50 SPLIT OF POWER SUPPLIES/SENSOR INPUTS
WE HACKED INTO THE RECEIVER'S "SIGNAL BEING RECEIVED" LED CIRCUIT SO THE FAILSAFE COULD MONITOR IT
DUAL RC RECEIVERS
DUAL POWER SUPPLIES

This is the current state of the avionics, after a drastic weight loss program and some more refinement on where redundancy is most required. There are only two user-programmable computers, the multicore Pi autopilot/companion computer and the original Arduino failsafe.

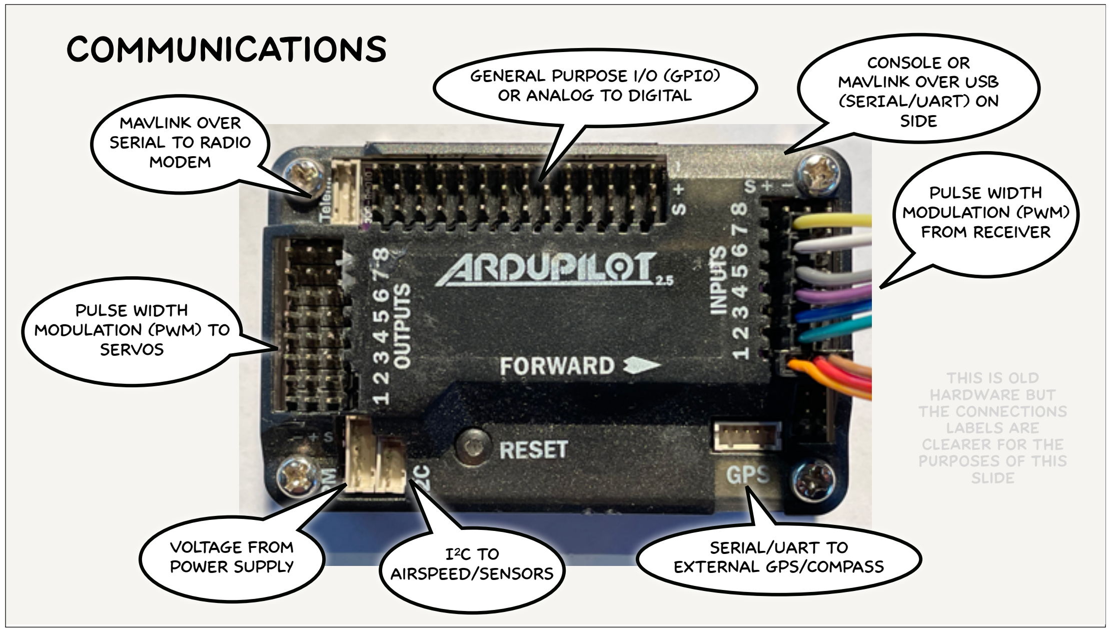
The failsafe is still the most critical redundant component. Under normal circumstances while receiving a heartbeat containing a location from the autopilot it leaves control of the UAV to the autopilot and the first RC receiver via the autopilot. At any point if the second RC receiver starts receiving it overrides the autopilot.

If the heartbeat position is outside the hard geofence (which cannot be changed during flight) or the heartbeat stops and there is no signal being received on the second RC receiver, the failsafe takes over control of the servos to send FTS inputs. If the failsafe itself looses power it leaves control with the autopilot and alarms will be going off.



Since neither the failsafe or the autopilot can make anything happen if the servos are unpowered, one of the changes we've made is to split the control surfaces between two power supplies so that if one fails there should be enough control left to limp home or engage the FTS.

Probably the key takeaway from the redundancy requirement is that it forces UAV software to be distributed. Speaking of distribution…


COMMUNICATIONS
GENERAL PURPOSE I/O (GPIO) OR ANALOG TO DIGITAL
CONSOLE OR MAVLINK OVER USB (SERIAL/UART) ON SIDE
MAVLINK OVER SERIAL TO RADIO MODEM
PULSE WIDTH MODULATION (PWM) FROM RECEIVER
PULSE WIDTH MODULATION (PWM) TO SERVOS
ARDUPILOT 2.5
OUTPUTS 1 2 3 4 5 6 7 8
INPUTS 1 2 3 4 5 6 7 8
FORWARD
RESET
GPS
THIS IS OLD HARDWARE BUT THE CONNECTIONS LABELS ARE CLEARER FOR THE PURPOSES OF THIS SLIDE
VOLTAGE FROM POWER SUPPLY
I²C TO AIRSPEED/SENSORS
SERIAL/UART TO EXTERNAL GPS/COMPASS

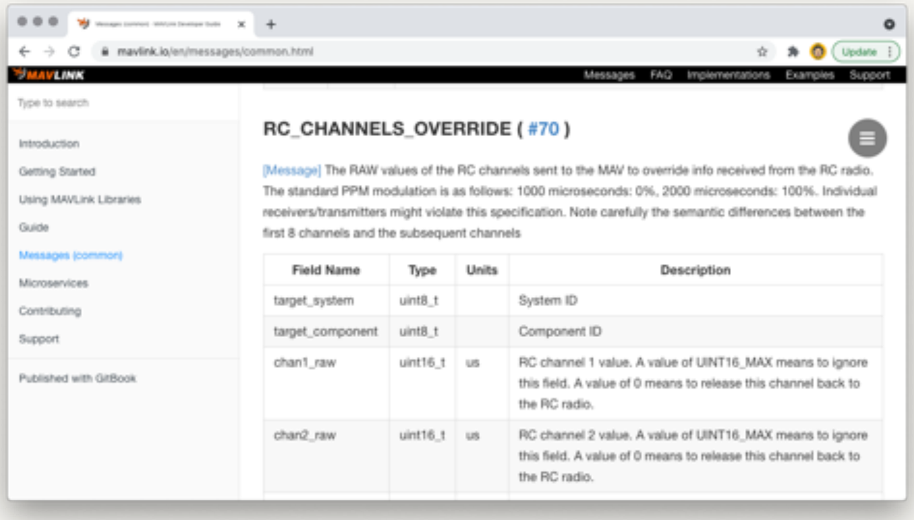
… how do all these things talk to each other?

* The most basic communication is through voltage levels, for instance the power supply battery voltage. To access it from your software you would set up the corresponding pin on your hardware to act as an analog to digital converter so that you can read the voltage as an integer.
* Simpler still are general purpose I/O pins that can be used as a boolean sensor or to switch things on and off - low power devices like LEDs can be powered by the pin, but more powerful devices will need to be switched via a remote switch called a relay. Arrays of GPIOs can be switched in parallel to control simple circuits like our servo demuxer, or flicked on and off according to some protocol - all the remaining communication methods are variations on this.
* Pulse width modulation is used to control a single servo - every 20ms you flick a pin on for between 1 and 2ms, and the length of the pulse indicates the desired servo position. A variation of this is Pulse Position Modulation (PPM) which round-robins PWM signals for multiple servos on a single wire, giving less redundancy but also a lot less weight because wires are heavy. Trick for young players, do not power your servos from your autopilot unless you want to experience the thrills of a voltage drop and autopilot mid-air reboot.
* I squared C is a popular, super simple two wire protocol for a microcontroller to communicate with over 1,000 peripheral devices (usually not microcontrollers in their own right) in a master-slave relationship - both our autopilots used I^{2}C for airspeed sensors.
* Serial communication is getting into the realm of normal software programming. Unsurprisingly it’s used for computer to computer

communication - this includes talking to more complex peripherals with embedded microcontrollers but particularly for air to ground comms because of limited bandwidth over radio modems and less complexity establishing or re-establishing connections. As for TCP/IP some people run PPP over serial, others might use mobile networks for air to ground. I mainly use the network at home for simulation, and then it's UDP messages rather than TCP sessions.

Of course there are drivers and libraries for all of these, but if you're new to microcontrollers it can take a while just to work out how and where each of them are used.

SOFTWARE - MAVLINK



RC_CHANNELS_OVERRIDE (#70)

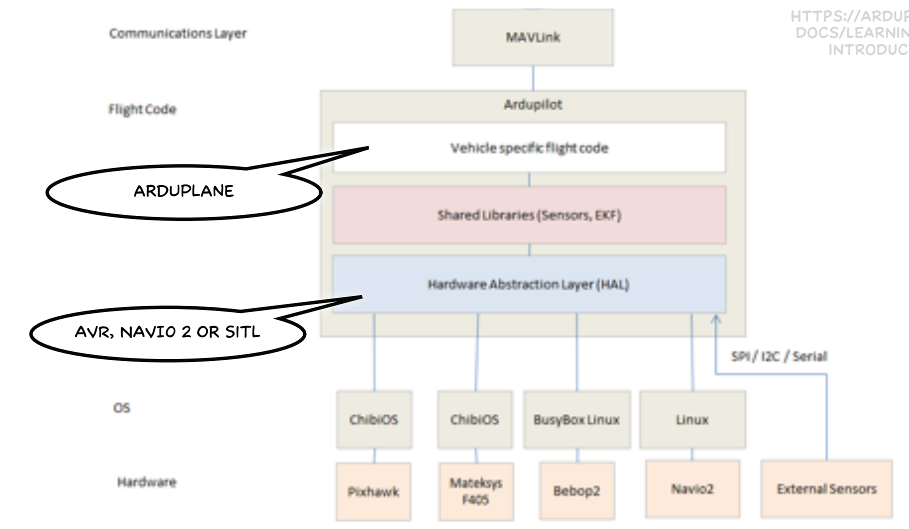
[Message] The RAW values of the RC channels sent to the MAV to override info received from the RC radio. The standard PPM modulation is as follows: 1000 microseconds: 0%, 2000 microseconds: 100%. Individual receivers/transmitters might violate this specification. Note carefully the semantic differences between the first 8 channels and the subsequent channels

Field Name	Type	Units	Description
target_system	uint8_t		System ID
target_component	uint8_t		Component ID
chan1_raw	uint16_t	us	RC channel 1 value. A value of UINT16_MAX means to ignore this field. A value of 0 means to release this channel back to the RC radio.
chan2_raw	uint16_t	us	RC channel 2 value. A value of UINT16_MAX means to ignore this field. A value of 0 means to release this channel back to the RC radio.

MAVLink (Micro Air Vehicle Link) is a communication protocol for publishing and subscribing to streams of data or sending commands between UAVs, ground stations and individual components. It is lightweight with only 8 to 27 bytes of overhead per message depending on the protocol version and whether signing is used. It can address up to 255 systems over noisy/high latency communication channels and has two levels of checksums at the message and message definition level.

Messages are defined in XML dialect files. The pymavlink project from Ardupilot provides code generators for C, C++, C#, Java, Javascript, Objective C, Lua, Typescript and Swift, and I've written one for Elixir available on hex.

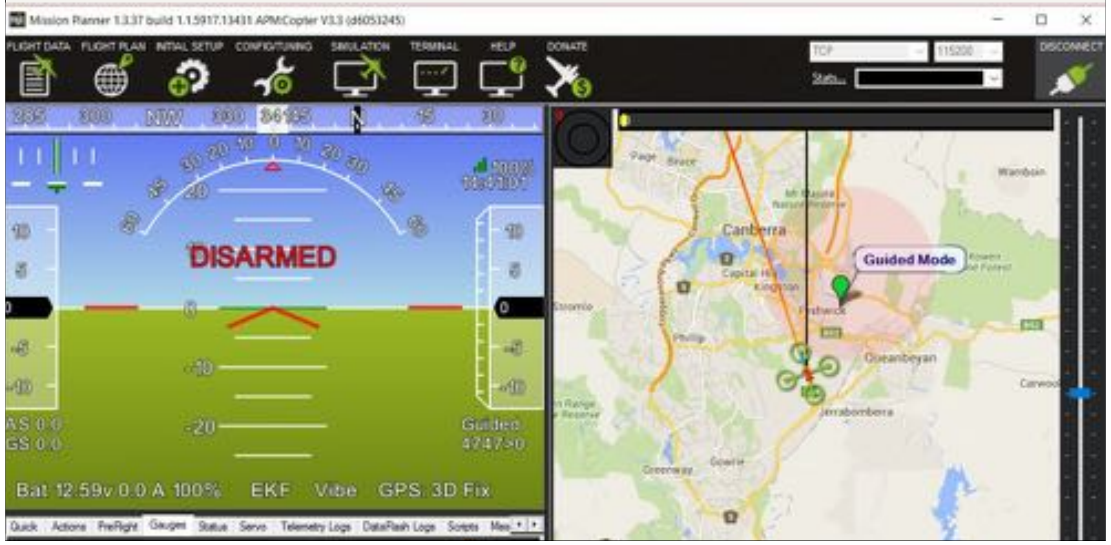
SOFTWARE - AUTOPILOT (ARDUPILOT)



Ardupilot is an umbrella project for open source autonomous vehicle software, including onboard autopilots for fixed wing planes, traditional helicopters and multicopters, rovers, submarines and antenna trackers (which sit at the ground station and point directional antennas at other vehicles for communications). The bulk of it is written in C++ and released under GPL3. The lead maintainer is Canberra-based Andrew Tridgell of RSync and Samba fame. Coincidentally Canberra UAV have been the team to beat for most of the history of the UAV Challenge.

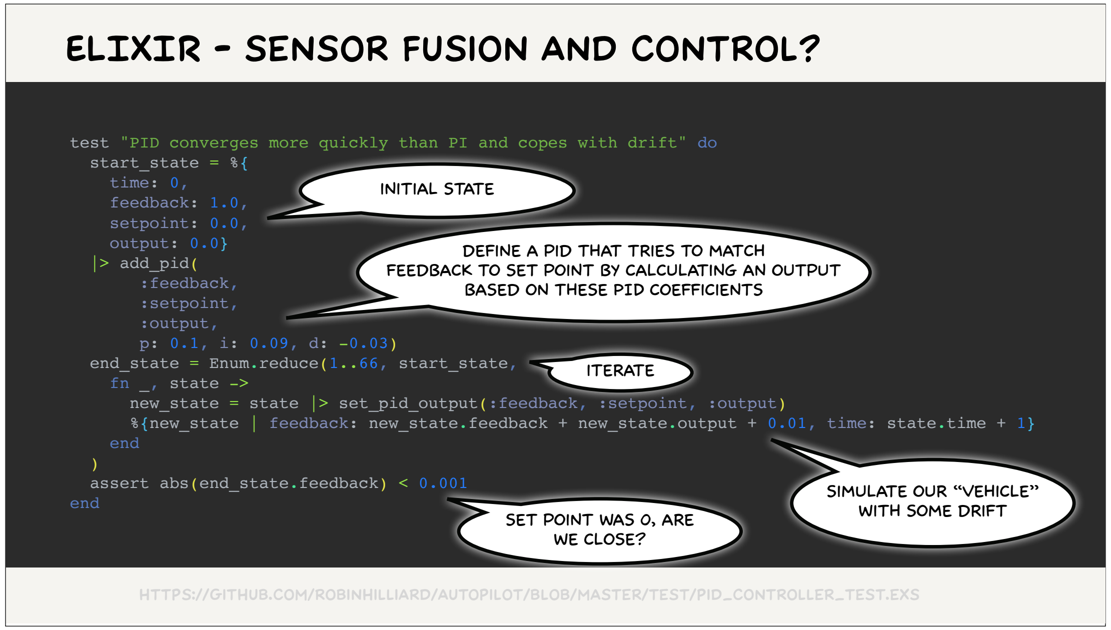
To grossly over simplify how it works, the main loop cycles the IMU/AHRS/Sensor Fusion/PID process and occasionally invokes loops that run at lower and lower frequencies to carry out other less real-time sensitive tasks. The code is nicely isolated from the various platforms Ardupilot runs on by the HAL, or Hardware Abstraction Layer. One of the most useful HAL implementations is the Software in the Loop or “SITL” which allows Ardupilot to run on my laptop hooked up to a vehicle simulator for testing.

SOFTWARE - GROUND STATION



A ground station provides a user interface for the UAV operators to calibrate and configure a vehicle, plan and upload a mission, arm and launch the UAV and monitor the mission progress. During flight users might redirect the UAV to another waypoint in the plan, tell it to circle at a point, return to base or hand over control to a pilot using a RC transmitter.

Ardupilot provides three ground stations, MavProxy APM Planner and Mission Planner. MavProxy is a modular CLI application written in Python, while APM Planner is a Mac and Windows desktop application and Mission Planner is Windows only.



ELIXIR - SENSOR FUSION AND CONTROL?

```
test "PID converges more quickly than PI and copes with drift" do
  start_state = %{
    time: 0,
    feedback: 1.0,
    setpoint: 0.0,
    output: 0.0}
  |> add_pid(
       :feedback,
       :setpoint,
       :output,
       p: 0.1, i: 0.09, d: -0.03)
  end_state = Enum.reduce(1..66, start_state,
    fn _, state ->
      new_state = state |> set_pid_output(:feedback, :setpoint, :output)
      %{new_state | feedback: new_state.feedback + new_state.output + 0.01, time: state.time + 1}
    end
  )
  assert abs(end_state.feedback) < 0.001
end
```

INITIAL STATE
DEFINE A PID THAT TRIES TO MATCH FEEDBACK TO SET POINT BY CALCULATING AN OUTPUT BASED ON THESE PID COEFFICIENTS
ITERATE
SIMULATE OUR “VEHICLE” WITH SOME DRIFT
SET POINT WAS 0, ARE WE CLOSE?

HTTPS://GITHUB.COM/ROBINHILLIARD/AUTOPILOT/BLOB/MASTER/TEST/PID_CONTROLLER_TEST.EXS

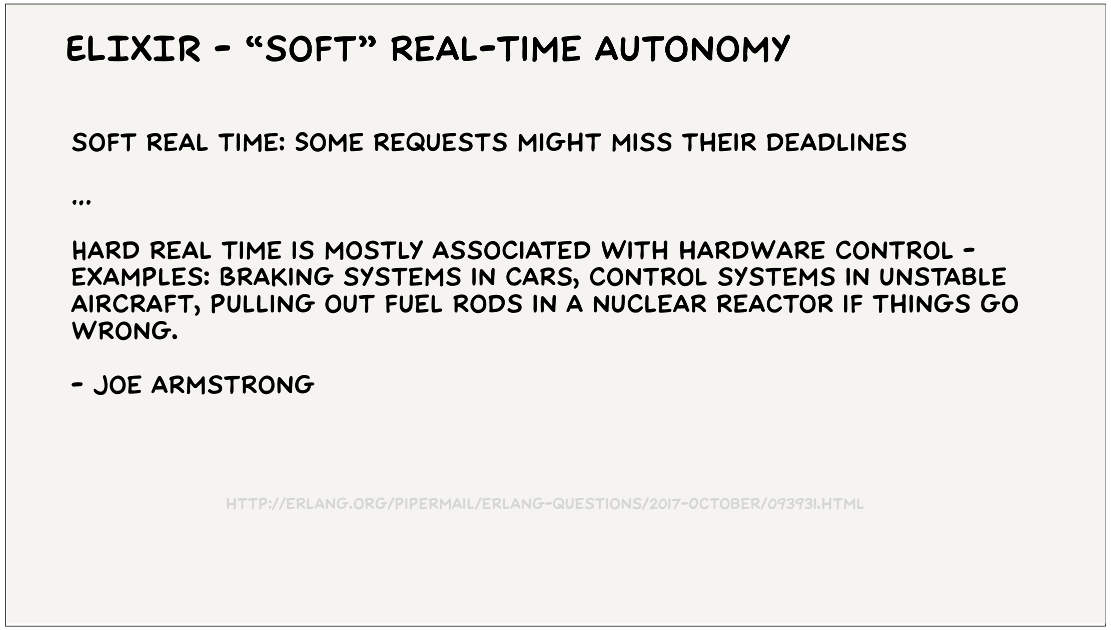
Where does Elixir fit in all of this?

A year back I demonstrated a simple autopilot using PIDs to control a light aircraft in an XPlane simulator at Elixir Sydney, so it can be done, but critically, that was a simulation: as geofencing goes, flying your aircraft in a separate reality is as good as it gets.

I’ve already explained how critical timing is to the sensor control loop. In the simulation we were easily exceeding the 20-30Hz required to control a general aviation full scale aircraft, but as we now know smaller, less stable aircraft like UAVs require higher frequencies, and why would you take the chance of trying to run software with hard safety related real-time requirements in a process that could be swapped out?

So no, I feel that the main loop of the autopilot should be running close to the metal in a language like C or Rust - think of it like a low level driver (literally) for the vehicle. In the same way a printer driver prints documents while hiding the PID loop controlling the print head, the autopilot takes altitude and heading set points and manipulates the controls to make it so.

If you’re running an autopilot on a multi-tasking OS you need to take the real-time requirement seriously - swapping your autopilot process out too often could make your vehicle uncontrollable. Navio provided a modified Linux kernel that prioritises interrupts so that the timer can trigger the Ardupilot sensor control loop reliably.



ELIXIR - "SOFT" REAL-TIME AUTONOMY

SOFT REAL TIME: SOME REQUESTS MIGHT MISS THEIR DEADLINES

...

HARD REAL TIME IS MOSTLY ASSOCIATED WITH HARDWARE CONTROL - EXAMPLES: BRAKING SYSTEMS IN CARS, CONTROL SYSTEMS IN UNSTABLE AIRCRAFT, PULLING OUT FUEL RODS IN A NUCLEAR REACTOR IF THINGS GO WRONG.

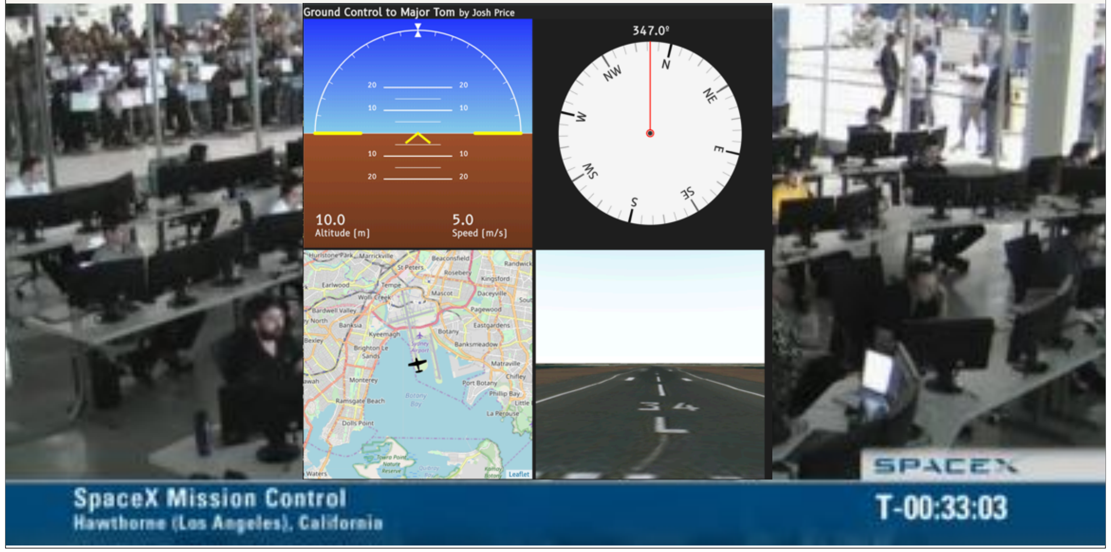
- JOE ARMSTRONG

HTTP://ERLANG.ORG/PIPERMAIL/ERLANG-QUESTIONS/2017-OCTOBER/093931.HTML

When we start thinking about the lower frequency autopilot loops responsible for things like dynamic flight planning things get more interesting. As I said earlier this is where the innovation and research is going on - tasks like flight path optimisation and constraint propagation can be undertaken concurrently in the background as long as they don't interrupt more important tasks, and the best-to-date flight plan being flown by the autopilot can be updated as better options become available.

BEAM/Erlang/Elixir is often described as being optimised for "soft" real-time software as encountered in the telephone exchange domain it was built for, which by happy coincidence is also highly suitable to web servers. I think this higher level autonomy code would (at the risk of annoying people perfectly happy writing it all in C) be a fantastic fit for Elixir.

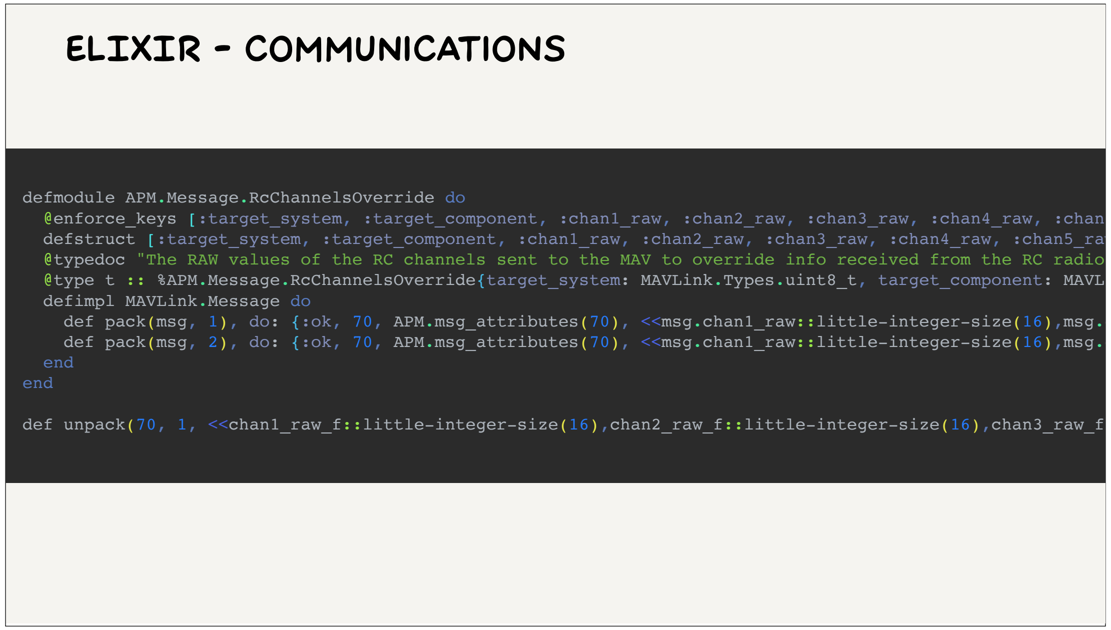
ELIXIR - PHOENIX LIVE VIEW GROUND STATION!


Ground Control to Major Tom by Josh Price
347.0°
10.0
Altitude [m]
5.0
Speed [m/s]
SPACEX
SpaceX Mission Control
Hawthorne (Los Angeles), California
T-00:33:03

Existing ground station software is improving but occasionally buggy, designed for a single user and a single point of failure for air to ground communications, even if redundant links are available. What if your ground station could be run as a cluster, with individual cluster members sharing their communication links, using distributed consensus algorithms for failover and allowing multiple users to log in and monitor particular aspects of the mission over a ground station local area network?

In other words, a perfect application for Elixir and Phoenix Live View.

* Josh Price wrote a flight simulator in live view that got runner-up in the 2019 Phoenix Frenzy competition, and we're hoping to marry this one day with MAVLink to make a bullet proof, multi-user ground station solution.



ELIXIR - COMMUNICATIONS

```
defmodule APM.Message.RcChannelsOverride do
  @enforce_keys [:target_system, :target_component, :chan1_raw, :chan2_raw, :chan3_raw, :chan4_raw, :chan
  defstruct [:target_system, :target_component, :chan1_raw, :chan2_raw, :chan3_raw, :chan4_raw, :chan5_ra
  @typedoc "The RAW values of the RC channels sent to the MAV to override info received from the RC radio
  @type t :: %APM.Message.RcChannelsOverride{target_system: MAVLink.Types.uint8_t, target_component: MAVL
  defimpl MAVLink.Message do
    def pack(msg, 1), do: {:ok, 70, APM.msg_attributes(70), <<msg.chan1_raw::little-integer-size(16),msg.
    def pack(msg, 2), do: {:ok, 70, APM.msg_attributes(70), <<msg.chan1_raw::little-integer-size(16),msg.
  end
end

def unpack(70, 1, <<chan1_raw_f::little-integer-size(16),chan2_raw_f::little-integer-size(16),chan3_raw_f
```

Elixir binary pattern matching, Dialyzer build-time type checks along with Erlang's XML and networking libraries makes working with MAVLink if not easy, then certainly extremely light weight and low boilerplate. My Elixir-MAVLink project provides a code generator to convert MAVLink dialect XML files into Elixir code and an easily configured, fully supervised application that lets your code speak to any MAVLink device over serial, UDP or TCP connections.

A VERY BRIEF DEMO

I could have spent the whole presentation on this demo but I felt it wouldn't add much to your understanding without the background information we've just been through. The goal is to show in the lightest possible way Elixir being used in UAV avionics.

I have a simulation of our UAV in the XPlane Flight Simulator. The Ardupilot Software in the Loop Hardware Abstraction Layer provides a way to connect Arduplane to this simulation, but I want to control the simulation from my radio transmitter. So I'll plug the radio receiver into one of my old Ardupilots to decode the PWM signals, and use Elixir to forward them to the Arduplane instance connected to the simulation.

FLY MY PRETTIES! QUESTIONS?

HTTPS://ARDUPILOT.ORG/

HTTPS://MAVLINK.IO/EN/

HTTPS://NAVIO2.EMLID.COM/

HTTPS://GITHUB.COM/BEAMUAV

@robinhilliard

robinhilliard

robin@rocketboots.com

Thanks and I hope I’ve inspired some of you to look further into UAV technologies.