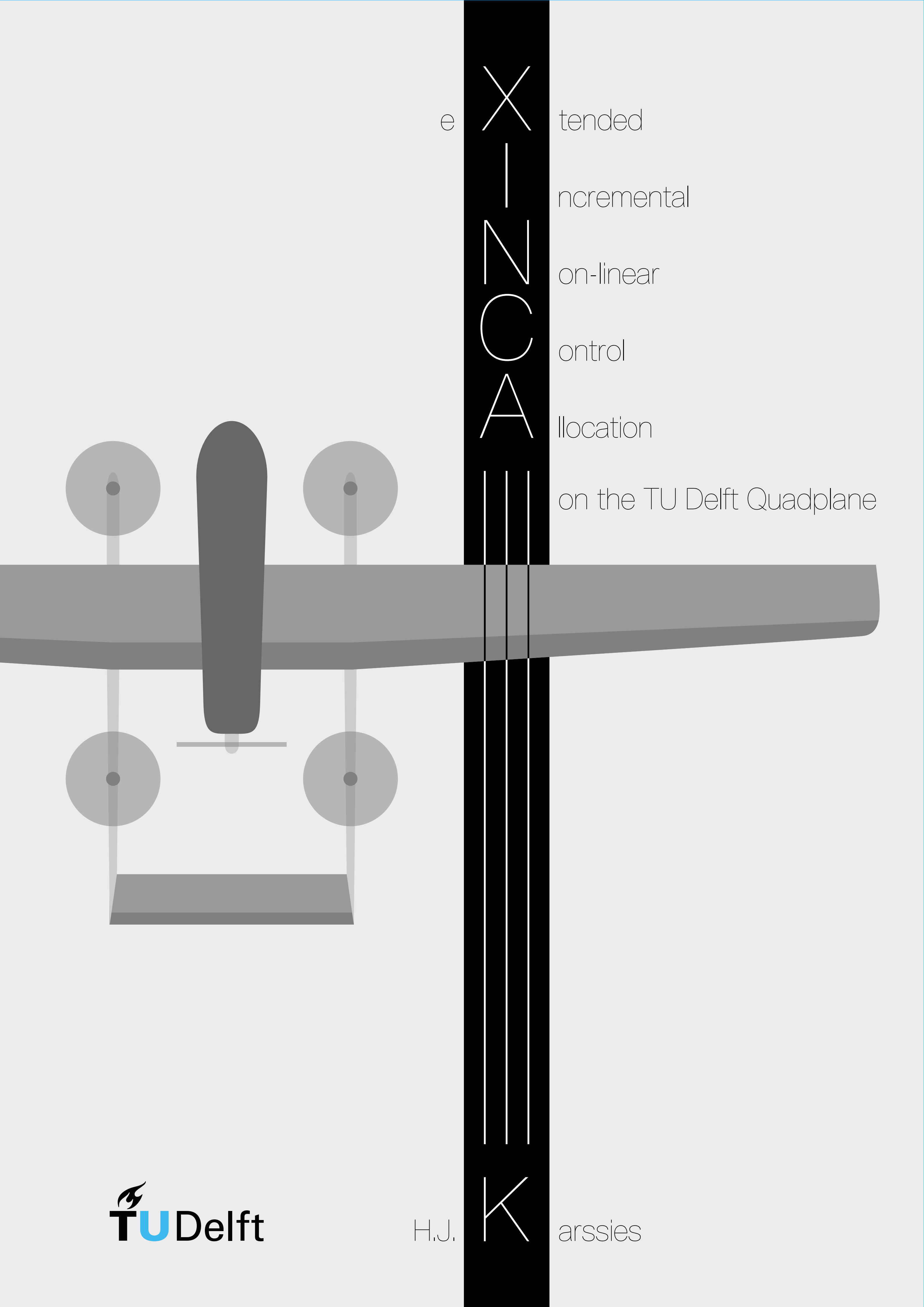

eXtended
Incremental
Non-linear
Control
Allocation
on the TU Delft Quadplane
H.J. Karssies
TU Delft

XINCA
on the TU Delft Quadplane

by

H.J. Karssies

to obtain the degree of Master of Science
at the Delft University of Technology.

Student number:	4479742			
Project duration:	September 1, 2018 – Oktober 30, 2020			
Thesis committee:	Prof. G.C.H.E. de Croon	TU Delft	AE Control & Operations	Chair
	Ir. C. de Wagter	TU Delft	AE Control & Operations	Supervisor
	Dr. E.J.J. Smeur	TU Delft	AE Control & Operations	Examiner
	Dr. S. Speretta	TU Delft	AE Space Engineering	External examiner

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

A friend once told me that me and the end of my thesis are like some asymptotic function and its asymptote. I was always getting closer to the end, without ever seeming to actually get there. Writing this thesis makes me (and him probably as well) very glad to prove him wrong. It certainly has been a struggle from time to time, but it seems I have actually made it in the end.

I started my thesis eagerly some amount of time in the past. With a slight lack of direction, I started to focus on many things that were somehow related to my thesis. I spent hours on making animations, worked meticulously on illustrations, and often got lost in other topics that later appeared to be slightly less relevant to my thesis than I initially thought. They did however gain me a lot of valuable knowledge an brought me a lot of joy during this phase of my thesis. Later in the process, I had to face myself a bit more. Working on a thesis while doing some tough last courses and working about two days a week did not always prove to be ideal. Then the Covid-19 pandemic kicked in right about when I wanted to start working on the actual quadplane. It was then especially that I was really glad my supervisor Christophe de Wagter was clearly doing his absolute best to enable me in continuing my work. Christophe, I had to learn throughout my thesis that my "zelluf doen" attitude sometimes prevented me to move forward. I remember many times that I was surprised how easily a meeting with you could get me going again. I think I should have asked for your insight more often, but want to sincerely thank you for all the times your help got me moving again. I owe the same thanks to Ewoud Smeur and Erik van der Horst, who have also spent a lot of time and effort to keep me going on crucial moments.

The last couple of months have certainly been the toughest. By the time I started writing my scientific article, I was working a full time job while spending my evenings and weekends on my thesis. These conditions certainly unabled me to pay my girlfriend the attention she deserves. Jacomijn, you have always been a formidable girlfriend, but I am truly impressed by the way you have supported me these last couple of months, and given me the space to do what I needed to do. I can't thank you enough for always looking at every new plot I made, or dealing with me when I was occasionally extremely grumpy. You have been the absolutely biggest reason why I have never felt the slightest bit unhappy during this tough process. I am very much looking forward to compensate for all the attention I have not given you the last couple of months.

I also want to thank some of my closest friends, some of which I have studied with intensively during my time in Delft. Siebert, Matti, Liset, Berna, you as well have been great contributors to both my motivation to keep going and much welcomed distractions between the hard work of the last couple of years. My biggest thanks here goes to my "vague acquaintance" Rogier, who has been there with me the whole way since the beginning of our bridging programme. Thanks for always finding the time to help me, give me your unsalted opinion, but most importantly provide me with an exceptional and extremely valuable friendship that will hopefully last a very long time.

Last but certainly not least, I want to thank the people that have known me the longest. Pap, mam, Stieneke, Elise, you know how much I value my family above almost anything in life. This is because you have always been such an endless and unconditional source of support and love, and still make me feel like coming home whenever I visit either one you. Thanks for letting me try to make you proud of me, without ever giving me the feeling I really have to.

Enough with the sentiments, it's almost time for me to celebrate finishing this chapter of my life. Again, thanks everyone! It's been a hell of a ride. Enjoy reading my thesis and all the best to every one of you!

Jan Karssies
Delft, September 2020

List of Figures

1 Overview of the quadplane's nine actuators 4
2 Simplified schematic UAV controller diagram 4
3 A schematic representation of an INCA controller 5
4 The Active Set Method performed on a hypothetical cost function with a two-dimensional input space 7
5 A schematic representation of a XINCA controller 8
6 Simulation of a vertical quadplane takeoff and landing 10
7 Simulation of a vertical quadplane takeoff and landing with actuator saturation 10
8 Simulation of forwards and backwards quadplane flight using both pitch increments and tail rotor inputs 11
9 The TU Delft Quadplane in the Cyberzoo for light tests 11
10 Flight data comparison of forward flight simulation with INDI and XINCA 11
11 Flight profile comparison of forward flight simulation with INDI and XINCA 12
12 Vertical quadplane takeoff and landing 13
13 Vertical quadplane takeoff and landing with actuator saturation 13
14 Forwards and backwards quadplane flight using both pitch increments and tail rotor inputs 13

2.1 UAV classification 21
2.2 Overview of the quadplane's nine actuators 23

3.1 A schematic representation of a controller with inner and outer control loops 26
3.2 A schematic representation of a PID controller 26
3.3 A schematic representation of an NDI controller 27
3.4 A schematic representation of an INDI controller 28
3.5 A schematic representation of an INCA controller 28

4.1 The Active Set Method performed on a cost function with a two-dimensional input space 35
4.2 The Active Set Method performed on a cost function with a two-dimensional input space 35

5.1 A schematic representation of an XINCA controller 37
5.2 Research planning 38

List of Tables

1 CPU load estimations for different inner and outer loop controllers with different numbers of INCA actuators . . . 12

A.1 Overview of simulations and flight experiments . . . 47

List of Abbreviations

CPU	Central Processing Unit
IMU	Inertial Measurement Unit
INCA	Incremental Non-linear Control Allocation
INDI	Incremental Non-linear Dynamic Inversion
NDI	Non-linear Dynamic Inversion
PID	Proportional-Integral-Derivative
PWM	Pulse Width Modulation
TU Delft	Technische Universiteit Delft (Delft University of Technology)
UAV	Unmanned Aerial Vehicle
VTOL	Vertical Take-Off and Landing
XINCA	Extended Incremental Non-linear Control Allocation

Nomenclature

Actuator inputs

δ	Control input
$\delta_{r_{lf}}$	Left front rotor control input
$\delta_{r_{rf}}$	Right front rotor control input
$\delta_{r_{lr}}$	Left rear rotor control input
$\delta_{r_{rr}}$	Right rear rotor control input
δ_{r_t}	Tail rotor control input
δ_a	Aileron control input
δ_{a_l}	Left aileron control input
δ_{a_r}	Right aileron control input
δ_{r_l}	Left ruddervator control input
δ_{r_r}	Right ruddervator control input
δ_{est}	Estimated actuator position
δ_{prev}	Previous estimated actuator position
δ_1	Control input for tail rotor
δ_2	Control input for all inputs but tail rotor
δ_r	Reference control input
δ_{r_1}	Reference control input for tail rotor
δ_{r_2}	Reference control input for all inputs but tail rotor
δ_p	Preference control input
δ_0	Current control input
δ_{min}	Minimum control input
δ_{max}	Maximum control input
$\dot{\delta}$	Control input rate
$\dot{\delta}_{max}$	Maximum control input rate
$\Delta\delta$	Incremental control input
$\Delta\delta_r$	Incremental reference control input
$\Delta\delta_{r_1}$	Incremental reference control input for tail rotor
$\Delta\delta_{r_2}$	Incremental reference control input for all inputs but tail rotor
$\Delta\delta_p$	Incremental preference control input
$\Delta\delta_{min}$	Minimum incremental control input
$\Delta\delta_{max}$	Maximum incremental control input
$\Delta\delta_k$	Incremental control input of current iteration
$\Delta\delta_{k-1}$	Incremental control input of previous iteration
$\Delta\delta_{viol_k}$	Saturated incremental control input of current iteration
$\Delta\delta_{viol_{k-1}}$	Saturated incremental control input of previous iteration

Matrices

$\mathbf{A}$	Constraint matrix
$\mathbf{A}_{act}$	Active set constraint matrix
$\mathbf{F}$	State matrix
$\mathbf{F}$	Matrix used to rewrite quadratic program
$\mathbf{G}$	Inertial matrix
$\mathbf{H}$	Actuator effectiveness matrix
$\mathbf{H_1}$	Actuator effectiveness matrix as a function of actuator positions
$\mathbf{H_2}$	Actuator effectiveness matrix as a function of actuator rates

$\mathbf{H}^+$	Generalized Moore-Penrose or psuedo-inverse of the actuator effectiveness matrix
$\mathbf{I}$	Identity matrix
$\mathbf{Q}$	Quadratic programming objective matrix
$\mathbf{W}$	Weighting matrix
$\mathbf{W}_\tau$	Control demand weighting matrix
$\mathbf{W}_\delta$	Actuator weighting matrix

Other Greek Symbols

α	First order actuator approximation coefficient
α	Maximum step factor
γ	Control objective scaling factor
λ	Lagrange multiplier
λ_{act}	Active set Lagrange multiplier
θ	Pitch angle
$\Delta\theta$	Pitch angle increment
ρ	Air density
τ	Time constant
τ	Achieved control
τ_c	Control demand
τ_{c_1}	Primary control demand
τ_{c_2}	Secondary control demand
ϕ	Roll angle
$\Delta\phi$	Roll angle increment
ω	Attitude
ω_e	Attitude error
ω_r	Reference attitude

Other Latin Symbols

b	Constraint vector
b_{act}	Active set constraint vector
b_{viol}	Saturated constraint vector
c	Quadratic programming objective vector
C_{L_α}	Lift coefficient
g	Gravitational acceleration
g	Vector used to rewrite quadratic program
H_{act}	Actuator transfer function
J	Cost function
K	Gain
K_D	Derivative gain
K_I	Integral gain
K_P	Proportional gain
k	Iteration number
m	Vehicle mass
N	Total number of iterations
$\Delta\dot{p}$	Roll acceleration increment
$\Delta\dot{q}$	Pitch acceleration increment
$\Delta\dot{r}$	Yaw acceleration increment
S	Wing surface area
T	Vertical thrust
ΔT	Vertical thrust increment
t	Time
Δt	Time step
u	Forward body velocity

υ	Virtual input
υ_e	Virtual input error
υ_f	Calculated current virtual input
υ_r	Reference virtual input
x	State
x_e	State error
x_r	Reference state
$\ddot{x}$	State acceleration
$\ddot{x}_r$	Reference state acceleration
$\ddot{x}_e$	Error in state acceleration
$\Delta\ddot{x}$	Longitudinal acceleration increment
z	Position in the z-direction
$\Delta\ddot{y}$	Lateral acceleration increment
$\ddot{z}$	Vertical acceleration
$\Delta\ddot{z}$	Vertical acceleration increment

Other Symbols

$\oslash$	Hadamard (element-wise) vector divider

Contents

Preface . . . iii
List of Figures . . . v
List of Tables . . . vii
List of Abbreviations . . . ix
Nomenclature . . . xi

I Part I: Scientific Research Paper 1
1 Introduction . . . 3
2 The TU Delft Quadplane . . . 4
3 INCA . . . 5
4 INCA Optimisation . . . 5
5 XINCA . . . 7
6 Implementation . . . 8
7 Flight Simulations . . . 10
8 Flight Experiments . . . 11
9 Conclusions and Recommendations . . . 13

II Part II: Preliminary Research Report 17
1 Abstract . . . 19
2 Introduction . . . 21
2.1 UAV Classification 21
2.2 The Quadplane 22
3 Control Allocation . . . 25
3.1 The PID controller 26
3.2 The NDI controller 27
3.3 The INDI controller 27
3.4 The INCA controller 28
4 INCA Optimisation . . . 31
4.1 Generalised Inverse 31
4.2 Redistributed Pseudo-Inverse 32
4.3 Quadratic programming 32
4.4 The Active Set Method 33
5 Further Research and Experiments . . . 37
6 Conclusion . . . 39
Bibliography . . . 41

III Part III: Appendices 45
A Overview of simulations and flight experiments . . . 47
B Paparazzi Airframe Configuration File . . . 49
C Source code of the INCA module . . . 55
D Source code of the XINCA module . . . 69

I

Part I: Scientific Research Paper

Extended Incremental Non-linear Control Allocation on the TU Delft Quadplane

H.J. Karssies - Delft University of Technology

Abstract - This research presents an implementation of a novel controller design on an over-actuated hybrid Unmanned Aerial Vehicle (UAV). This platform is a hybrid between a conventional quadcopter and a fixed-wing aircraft. Its inner loop is controlled by an existing but modified control method called Incremental Non-linear Control Allocation or INCA. This controller deals with the platform's control allocation problem by minimising a set of objective functions with a method known as the Active Set Method and avoids actuator saturation. For the vehicle's outer loop, a novel extension to INCA is presented, called Extended INCA or XINCA. This method optimises one of the physical actuator's command and the angular control demands fed to the vehicle's inner loop, based on linear reference accelerations. It does so while adapting to varying flight phases, conditions and vehicle states, and taking into account the aerodynamic properties of the main wing. XINCA has low dependence on accurate vehicle models and requires configuration using only several optimisation parameters. Both flight simulations and experimental flights are performed to prove the performance of both controllers.

1 Introduction

In the first two decades of this century, Unmanned Arial Vehicle or UAVs have gained a tremendous amount of popularity. Not only have they proven to be valuable research platforms and entertaining toys, they have also found many other applications in fields like defence [1], surveillance [2], medical assistance [3], transportation of both goods and humans [4], agriculture [5], inspection [6], mapping [7], and many others. The rising demand in UAVs stimulates engineers and researchers to push the boundaries of unmanned aviation, and often come up with the most innovative of ideas.

Some challenges that are often faced in UAV design are endurance, reliability, versatility and affordability. Existing solutions often perform well on some but not all of these aspects. Fixed wing aircraft like the ones by Daibing et al. [8], Palermo and Vos [9] and Lee et al. [10] for instance master endurance as a result of the passive wing-induced lift that keeps them airborne. Rotorcraft on the other hand, like designs by Zhiqiang et al. [11], Luukkonen [12] and Smeur et al. [13], are much more versatile since they are able to hover, takeoff and land vertically. They are also inexpensive to produce, mechanically simple and easy to control. Their powered generation of lift however severely limits their endurance, and designs like the conventional quadcopter typically have multiple single points of failure. It is therefore that many researchers have come up with hybrid platforms, that aim to combine the best of different worlds.

Some examples of hybrid platforms include tilt rotor/wing UAVs, tail sitters, transformable UAVs and quadplanes. Tilt rotor/wing UAVs like designs by Apkarian [14] and Takeuchi et al. [15] mechanically change the orientation of their propulsion units in order to either generate lift during vertical take off and landing, or horizontal thrust while flying horizontally with wing induced lift. Similarly, tail sitters as discussed by De Wagter and Smeur [16] and Argyle et al. [17] change the orientation of the entire vehicle during vertical take off and landing, before slowly rotating back to their original orientation for horizontal flight. This reduces the mechanical complexity of the system, resulting in a more reliable, lighter and cheaper platform, albeit at the cost of a sensitivity to wind gusts. A completely different class of hybrid UAVs are the ones that are transformable like the one designed by Shaiful et al. [18]. By changing the configuration of the entire vehicle, they can transform between very different types of UAVs, like for instance a monocopter and a fixed wing aircraft.

Lastly, a common class of hybrid UAVs is formed by quadplanes, like the one used as an experimental platform for this research. Earlier designs include those by Gunarathna and Munasinghe [19], Zhang et al. [20], Orbea et al. [21], Tielin et al. [22] and Flores and Lozano [23]. The quadplane has a static configuration with both upward facing rotors for vertical take off and landing, and fixed wings with a horizontal propulsion unit for horizontal flight. Despite the added weight of flight phase specific actuators, its

mechanical simplicity makes this versatile and enduring vehicle a promising research platform.

Making such a Quadplane fly as efficiently and safely as possible poses a number of challenges. These include dealing with large flight envelopes, over-actuation, its non-linear nature, and its sensitivity to wind gusts. The quadplane used for this research and its control challenges are described in Section 2. An existing control method called INCA is discussed in Section 3, and its optimisation methods in Section 4. A proposed extension of this control method, called XINCA, is presented in Section 5. The implementation of the INCA and XINCA controllers on the TU Delft Quadplane is shown in Section 6, and Sections 7 and 8 respectively present results from simulations and test flights performed using this novel control method. Lastly, Section 9 discusses the conclusions and recommendations of this research.

2 The TU Delft Quadplane

As mentioned earlier, the quadplane is a hybrid of a fixed wing aircraft and a quadcopter. A conventional example of a quadplane is the one used for this research, the *TU Delft Quadplane*. A schematic representation of this platform is shown in Figure 1. It shows the quadplane's nine actuators: four upward facing rotors that could be considered as the *quadcopter actuator set*, and four control surfaces and a tail rotor that could be considered the *fixed wing actuator set*. Having actuator sets to serve both vertical and horizontal flight separately, quadplanes are considered *over-actuated*. Literature shows that this over-actuation is often dealt with by using only one actuator set during specific flight phases, and only briefly combining them during a transition phase between vertical and horizontal flight [19, 20, 21, 22].

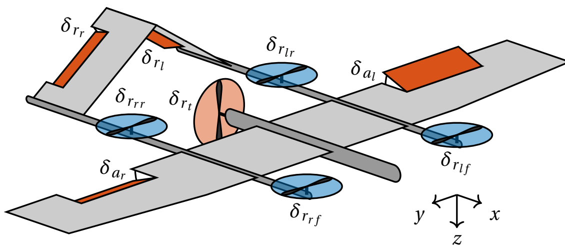


Figure 1: Overview of the quadplane's nine actuators
(= quadcopter actuator set, = fixed wing actuator set)

UAV controllers often consist of cascaded outer and inner loops, as shown in the simplified schematic representation in Figure 2. The outer loop, sometimes also called the position or guidance loop, measures the vehicle's deviation from its reference position, and outputs a reference attitude needed to decrease this error. The inner loop, which is also known as the attitude or stabilisation loop, in turn determines the error between this reference attitude and the actual attitude, and uses that to allocate control to suitable actuators. This allocation is quite straightforward when the vehicle is not over-actuated, like a quadplane when only one actuator set is taken into account. This is simply because a moment around any one of the principal axes can only be achieved by one specific combination of actuator inputs.

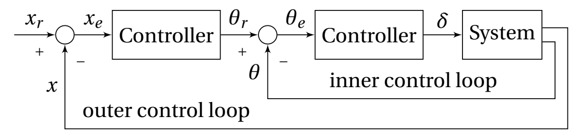


Figure 2: Simplified schematic UAV controller diagram
(x = position, θ = attitude, δ = system input)

This research however hypothesises that quadplanes could fly more efficiently when continuously assessing each actuator's suitability to satisfy a certain control demand. This assessment takes into account each actuator's effectiveness based on the system's states, but could also penalise large deviations from preferred actuator positions. Such an optimisation problem is known as a control allocation problem. There are two main advantages of a well designed control allocation algorithm. The first is that it can minimise the control effort of a UAV, potentially resulting in more efficient flight and enhanced flight endurance. The other advantage is that when certain actuators saturate, it can allocate control to other actuators in order to still satisfy a given control demand, resulting in safer and more reliable flight. The control allocation method used in this research is called Incremental Non-linear Control Allocation or INCA, which solves the inner loop's control allocation optimisation problem and is presented in Chapter 3.

Another challenge in controlling quadplanes is caused by the fundamentally different outer loop dynamics of the quadplane during different flight phases. When flying as a quadcopter for instance, a change in pitch angle causes the quadplane to accelerate in a longitudinal direction. When flying as a fixed-wing aircraft however, a change in pitch will cause the quadplane to either climb or descent. Furthermore, the quadplane is over-actuated in its outer loop as well as its inner loop, since it can control a positive forward acceleration during hovering with both its pitch angle as well as its tail rotor. The latter is often preferable, since negative pitching manoeuvres might introduce an undesirable negative wing-induced lift. A positive backwards acceleration however is only achievable by pitching the quadplane backwards. To address the challenges named above,

an extension of the INCA controller is presented in Chapter 5, which performs an outer loop optimisation similar to the INCA inner loop optimisation. This method is called Extended Incremental Nonlinear Control Allocation, or XINCA for short.

3 INCA

Incremental Non-linear Control Allocation, or INCA for short, is very promising control allocation algorithm. It has already theoretically been demonstrated on over-actuated vehicles like the Lockheed Martin Innovative Control Effector aircraft by Stolk [24]. Smeur et al. [25] have proven the control method to be effective in actual flight on non-over-actuated quadcopters. The architecture of INCA augments a method called Non-linear Dynamic Inversion, or NDI. NDI measures a vehicle's states, and uses an accurate model to predict angular and possibly linear accelerations as a result of these states. Their difference with the vehicle's desired accelerations is then used to calculate appropriate control inputs using reliable actuator models. A successful example of an implementation of NDI is the work by Horn [26].

However effective, NDI highly relies on detailed and accurate models of the vehicle it controls. A variation on this approach provides a solution to this problem, and is called Incremental Non-linear Dynamic Inversion, or INDI [25]. Instead of using a vehicle model to predict its angular and linear accelerations as a result of its states, it uses inertial measurement data to simply observe these accelerations, resulting in a controller that does not require accurate vehicle models. Also in contrast to NDI, the measurements used by INDI include all achieved control already, including the actuator-induced accelerations of the vehicle, but also effects of external forces caused for instance by wind gusts. This results in a desired *incremental* control demand instead of a total actuator control demand, only containing yet to be achieved control. As a consequence, the control effectiveness needs not be as accurate as earlier, since the controller will compensate for any unexpected effects of the actuators. The controller is also less sensitive to external influences. An example where INDI has been proven successfully in quadcopter flight is presented by Höppener [27].

Both NDI and INDI invert actuator effectiveness models in order to calculate appropriate actuator commands. When dealing with over-actuated UAVs however, it becomes inherently impossible to derive appropriate actuator commands by simply inverting these actuator effectiveness models. This is due to the fact that any calculated actuator command solution is no longer singular, and for it there exist infinite other solutions. INCA deals with this by expressing this control allocation problem as an optimisation problem, that needs to be solved by minimising a certain cost function. While doing so, it can take into account actuator constraints, preventing actuator saturation. A schematic representation of INCA is shown in Figure 3. Like an INDI controller, INCA uses the difference between desired accelerations and inertial measurements to determine an incremental control demand, also known as the virtual input to the INCA optimisation. The optimisation scheme then calculates an optimal actuator increment to satisfy the control demand as well as possible, based on the actuators' effectiveness at the vehicles current state. Note that while the rotors' effectiveness is relatively constant, the effectiveness of any control surfaces included in the INCA optimisation is proportional to the square of the vehicle's true air speed. The effectiveness of these actuators should therefore be re-calculated at each iteration of the INCA optimisation. The optimisation method itself is further elaborated in Section 4.

4 INCA Optimisation

Let $\mathbf{H}$ be a matrix containing the linearised effectiveness of all actuators, and τ_c the control demand that will be used as virtual input to the INCA optimisation. An unconstrained control command increment $\Delta\delta$ should then always satisfy the following equation:

$$\mathbf{H}\Delta\delta = \tau_c \tag{1}$$

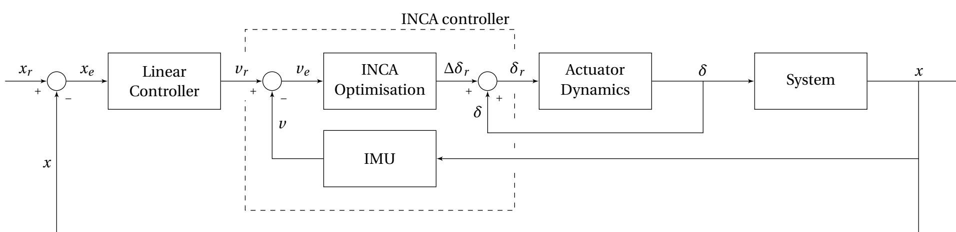


Figure 3: A schematic representation of an INCA controller
(x = state vector, ν = virtual input, δ = control input vector)

When this increment is constrained by actuator limits however, an error between the control demand and the achieved control might occur, but should still be aimed to be minimised. When also minimising control effort, i.e., the difference between actual actuator increments $\Delta\delta$ and preferred actuator increments $\Delta\delta_p$, an objective function could be written in the following form:

$$\min_{\Delta\delta} \|\gamma \mathbf{W}_\tau (\mathbf{H}\Delta\delta - \tau_c)\|_2 + \|\mathbf{W}_\delta(\Delta\delta_p - \Delta\delta)\|_2 \tag{2a}$$

$$\text{subject to } \Delta\delta_{min} \le \Delta\delta \le \Delta\delta_{max} \text{ and } \dot{\delta} \le \dot{\delta}_{max} \tag{2b}$$

where $\mathbf{W}_\tau$ and $\mathbf{W}_\delta$ are weighting matrices to prioritise certain control demands and actuators over others, and γ is a constant that prioritises one sub-objective over the other. This type of objective function is called a *Quadratic Program*, and can include as many separate sub-objectives as needed. Quadratic Programming is often used for Control Allocation problems. Härkegård [28] presents it as a suitable method, and proves that it does indeed provide automatic redistribution of control in case of actuator saturation. Stolk [24] and Höppener [27] both apply it, on a modern fighter jet and a quadcopter UAV respectively. For easier processing, the objective function is often rewritten to a standardised quadratic form, with which many solvers can easily work:

$$\min_{\Delta\delta} \Delta\delta^T \mathbf{Q}\Delta\delta + c^T\Delta\delta \tag{3a}$$

$$\text{subject to } \mathbf{A}\Delta\delta \le b \tag{3b}$$

where $\mathbf{Q} = \mathbf{F}^T\mathbf{F}$, $c = 2\mathbf{F}^T g$,

$$\mathbf{F} = \begin{pmatrix} \gamma\mathbf{W}_\tau\mathbf{H} \\ \mathbf{W}_\delta \end{pmatrix},\ g = \begin{pmatrix} \gamma\mathbf{W}_\tau\tau_c \\ \mathbf{W}_\delta\Delta\delta_p \end{pmatrix},$$

$$\mathbf{A} = \begin{pmatrix} \mathbf{I} \\ -\mathbf{I} \end{pmatrix} \text{ and}$$

$$b = \begin{pmatrix} \min(\delta_{max} - \delta_0, \dot{\delta}_{max}\Delta t) \\ -\max(\delta_{min} - \delta_0, -\dot{\delta}_{max}\Delta t) \end{pmatrix}$$

According to Gavin and Scruggs [29], when the inequality constraints are treated as equality constraints ($\mathbf{A} = b$ instead of $\mathbf{A} \le b$), the solution to the optimisation problem is given by the following linear system, as long as $\mathbf{Q}$ is a positive definite matrix and $\mathbf{A}$ has full row rank [30]:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \lambda \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix} \tag{4}$$

where λ is also know as the vector containing the Lagrange multipliers. From this linear system, explicit solutions for both the optimal input increment $\Delta\delta$ and Lagrange multipliers λ can be derived algebraically to:

$$\Delta\delta = -\mathbf{Q}^{-1}(\mathbf{A}^T\lambda + c) \tag{5a}$$

$$\text{where } \lambda = -(\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^T)^{-1}(\mathbf{A}\mathbf{Q}^{-1}c + b) \tag{5b}$$

The values of the Lagrange multipliers are used later to determine what constraints to release during the optimisation process, and whether or not the solution has already reached its optimum.

Since the calculation of UAV control demands typically needs to be performed several hundred times per second, the optimisation used in an INCA controller needs to be as efficient as possible. Based on control allocation research performed by Stolk [24] and Höppener [27], the optimisation method selected for this research is the *Active Set Method*. This method is presented to require similar amounts of computing power as competing methods do, like the Redistributed Pseudo-Inverse method and the Fixed-Point algorithm, yet with more accurate solutions. The method also promises to scale efficiently with larger amounts of actuators, which is validated in Section 8. A detailed description of the Active Set Method as provided by Harkegard [31] is summarised below:

Step 1:

Choose a feasible starting point

Step 2:

Determine the *active set* of constraints, i.e. all constraints at which a control command saturates. Redefine the optimisation problem using only the active constraints as equality constraints.

Step 3:

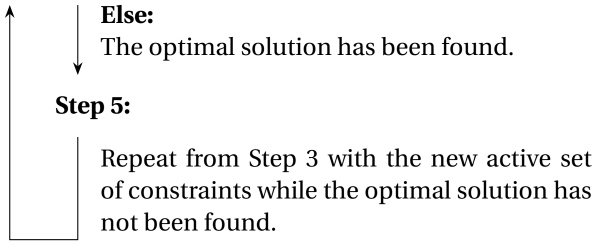
Calculate the Lagrange multipliers and solution to the redefined problem using Equations 5a and 5b.

Step 4:

If the solution is infeasible:
Correct the solution by taking the maximum relative step from the previous to the new solution without losing feasibility and determine the new active set of constraints.

Else if not all $\lambda \ge 0$:
Release the constraint corresponding to the most negative value in λ from the active set of constraints.

(Steps 3–4 repeated for $k = 1, 2, \ldots, N$)



Else:
The optimal solution has been found.

Step 5:

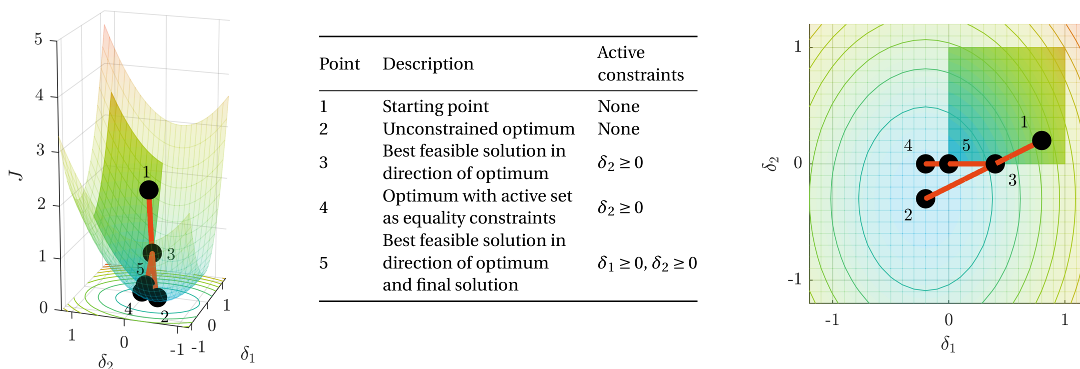
Repeat from Step 3 with the new active set of constraints while the optimal solution has not been found.

Figure 4 shows an illustrative example of the process described above for a hypothetical optimisation problem with a constrained two-dimensional input space. Choosing a suitable starting point for the Active Set Method has a significant effect on the solver's efficiency. In control allocation however, a smart starting point is always at hand, since each solution is likely to be in the neighbourhood of the solution of the previous time step. The Active Set Method has a relatively low computational cost [31], and the solver's solution during each iteration always progresses towards the final solution of that time step. This means that chances are small that the solver will produce a most unsuitable solution if cut off shortly. This results in the Active Set Method being very suitable for control allocation applications.

5 XINCA

One problem mentioned in Section 2 is the fundamentally different dynamics of a quadplane during different flight phases. Vertical acceleration for instance is achieved by increasing or decreasing vertical thrust during hovering, but could be achieved more efficiently during horizontal flight by pitching up or down. Another complexity in quadplane control is that longitudinal acceleration during hovering is typically achieved by pitching forwards an backwards, while *forward* longitudinal acceleration could likewise be achieved by use of the quadplane's tail rotor. Furthermore, when pitching forward in order to achieve forward acceleration, an undesirable downforce could be induced by the wing's negative angle of attack, resulting in inefficient flight. To simplify matters, hybrid UAVs like quadplanes are often controlled in either a vertical, horizontal or transitional flight mode. Separating these flight modes however often results in sub-optimal flight control, not always using the most effective actuators at hand nor making use of redundant actuators in case of actuator saturation. To solve the problems mentioned above, an extension to the INCA controller is proposed in this research, called Extended Incremental Non-linear Control Allocation or XINCA for short.

A XINCA controller is similar to an INCA controller, except that its optimisation process is performed in the system's outer control loop. As seen in Figure 5, a linear controller observes the error of the quadplane's position, and translates this error into linear reference accelerations. The error between these reference accelerations and measured accelerations provides the control demand to the XINCA optimisation block. Like the INCA optimisation, the XINCA optimisation possesses several constrained actuators to achieve this control demand with, albeit these XINCA actuators do not only include physical actuators of the platform, but also some of its attitude angles and its vertical thrust command. In the case of this research, the XINCA output includes the tail rotor command, the vertical thrust command, and the vehicle's pitch and roll commands. The tail rotor command is directly fed to the tail rotor itself. The thrust command and two attitude angle commands serve as input for the inner loop's INCA optimisation. The XINCA optimisation can, like the INCA optimisation, be performed with the Active Set Method, again allowing an additional sub-objective to minimise the difference between the XINCA actuators and their preferred positions.



Point	Description	Active constraints
1	Starting point	None
2	Unconstrained optimum	None
3	Best feasible solution in direction of optimum	$\delta_2 \geq 0$
4	Optimum with active set as equality constraints	$\delta_2 \geq 0$
5	Best feasible solution in direction of optimum and final solution	$\delta_1 \geq 0, \delta_2 \geq 0$

Figure 4: The Active Set Method performed on a hypothetical cost function J with a two-dimensional input space (Constraints: $0 \leq \delta_1 \leq 1$ and $0 \leq \delta_2 \leq 1$, starting point: $(\delta_1, \delta_2) = (0.8, 0.2)$)

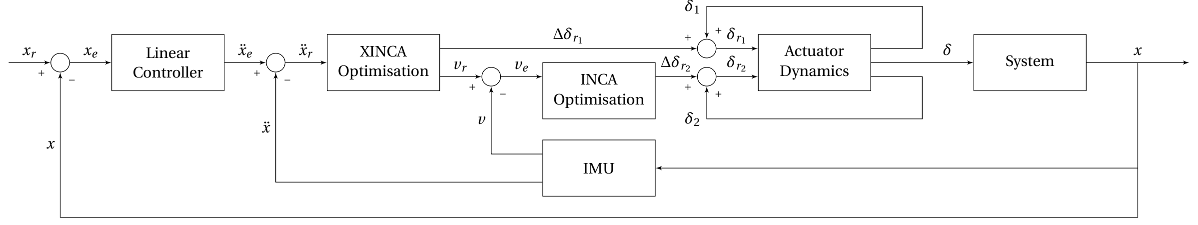


Figure 5: A schematic representation of a XINCA controller
(x = state vector, v = virtual input, δ = control input vector)

The effectiveness of the XINCA actuators are also highly dependent on the aircraft's states. They therefore need to be re-assessed at every iteration in order to ensure suitable outputs. As a result of this novel controller method, an autonomous outer loop controller or human operator only needs to control the UAV's linear accelerations, without ever having to worry about varying flight conditions or flight modes it might be in.

6 Implementation

The TU Delft Quadplane's implementation of XINCA is done by use of the open-source drone hardware and software platform Paparazzi UAV [32]. The quadplane itself makes use of a *Lisa/MX autopilot* board. Since this board can control a maximum of eight actuators, the quadplane's two ailerons share one control command, making them respond symmetrically yet in opposite direction. Note that this reduction in actuator commands benefits the computational cost of the INCA optimisation.

6.1 INCA

The INCA module in Paparazzi UAV is based on an existing INDI module from a research by Smeur et al. [25], which already makes use of an optimisation module from another research, also by Smeur et al. [13]. It is extended in order to include seven of the quadplane's eight actuators, and scale the effectiveness of the three actuators that are control surfaces, i.e. two separate ruddervators and and the combined ailerons. The achieved control is calculated as follows:

$$\begin{bmatrix}\Delta\dot{p} & \Delta\dot{q} & \Delta\dot{r} & \Delta\ddot{z}\end{bmatrix}^T = \mathbf{H}\Delta\delta \tag{6}$$

where $\delta = \begin{bmatrix}\delta_{r_{lf}} & \delta_{r_{rf}} & \delta_{r_{rr}} & \delta_{r_{lr}} & \delta_a & \delta_{r_l} & \delta_{r_r}\end{bmatrix}^T$

The control effectiveness matrix $\mathbf{H}$ is separated into two parts. $\mathbf{H_1}$ accounts for increments in actuator inputs, and $\mathbf{H_2}$ accounts for counter torque effects during the spin-up of the upwards facing rotors, such that:

$$\mathbf{H} = \mathbf{H_1} + \Delta t \mathbf{H_2} \tag{7}$$

The actuator effectiveness matrices' units are either rads^{-2}PPRZ^{-1} or ms^{-2}PPRZ^{-1}, where PPRZ stands for Paparazzi actuator units ranging from -9600 for bi-directional or 0 for mono-directional actuators to 9600. To prove INCA's ability to handle inaccurate actuator models because of its incremental nature, only a simple approximation of the actuators' effectiveness is used to control the quadplane. This approximation is based on simple calculations using estimations of the quadplane's inertial properties and its actuators' positions relative to its centre of gravity. The resulting actuator effectiveness matrices are:

$$\mathbf{H_1} = 10^{-3} \cdot \begin{array}{c} \\ \end{array} \overset{\begin{matrix}\delta_{r_{lf}} & \delta_{r_{rf}} & \delta_{r_{rr}} & \delta_{r_{lr}} & \delta_a & \delta_{r_l} & \delta_{r_r}\end{matrix}}{\begin{bmatrix} 11 & -11 & -11 & 11 & 0.15u^2 & 0 & 0 \\ 9 & 9 & -9 & -9 & 0 & 0.11u^2 & -0.11u^2 \\ -0.6 & 0.6 & -0.6 & 0.6 & 0 & -0.03u^2 & -0.03u^2 \\ -0.8 & -0.8 & -0.8 & -0.8 & 0 & 0 & 0 \end{bmatrix}} \begin{matrix}\Delta\dot{p} \\ \Delta\dot{q} \\ \Delta\dot{r} \\ \Delta\ddot{z}\end{matrix} \tag{8a}$$

$$\mathbf{H_1} = 10^{-3} \cdot \overset{\begin{matrix}\delta_{r_{lf}} & \delta_{r_{rf}} & \delta_{r_{rr}} & \delta_{r_{lr}} & \delta_a & \delta_{r_l} & \delta_{r_r}\end{matrix}}{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -55 & 55 & -55 & 55 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}} \begin{matrix}\Delta\dot{p} \\ \Delta\dot{q} \\ \Delta\dot{r} \\ \Delta\ddot{z}\end{matrix} \tag{8b}$$

where u ideally represents the true airspeed over the control surfaces, which in this research is simplified by the substitution of the forward body velocity, since tests are performed in an indoor environment without wind. Negative values of this velocity are replaced by zero.

Since the quadplane's actuators do not provide any form of feedback, an estimation of the current actuator positions needs to be performed for each time step. This is done by a first order approximation with a certain time constant τ:

$$H_{act} = \frac{K}{\tau s + 1} \tag{9}$$

Each current actuator position is estimated as follows:

$$\delta_{est} = \delta_{prev} + \alpha(\delta - \delta_{prev}) \quad (10)$$
$$\text{where } \alpha = 1 - e^{-\tau \Delta t}$$

In this research, the time constants used for the four upwards facing rotors are all 29 s^{-1}, based on actuator response measurements. For the control surfaces, an estimation of 100 s^{-1} is used. Looking at Equations 2a and 2b: the optimisation parameters are chosen as follows:

$$\mathbf{W}_\tau = diag\begin{bmatrix}100 & 100 & 1 & 1000\end{bmatrix}$$
$$\mathbf{W}_\delta = diag\begin{bmatrix}10 & 10 & 10 & 10 & 1 & 1 & 1\end{bmatrix}$$
$$\gamma = 10000$$
$$\delta_p = diag\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

The values of $\mathbf{W}_\tau$ prioritise pitch and roll and especially thrust commands over the yaw command, $\mathbf{W}_\delta$ penalises the use of rotors over the use of control surfaces, and γ prioritises achieving the control demand over minimising control effort. The actuator limits are set to either 0 and 9600 for rotors or -9600 and 9600 for control surfaces, again expressed in PPRZ units. The actuator rate limits are being discarded.

6.2 XINCA

The XINCA controller works in a similar manner as the INCA controller, and is based on an existing outer loop INDI module by Smeur et al. [33, 34]. This existing module uses the vertical thrust vector to control the quadplane's position, by either changing this thrust itself or changing its orientation by pitch or roll increments. It is augmented by including a tail rotor command as its fourth actuator. The achieved control is then calculated as follows:

$$\begin{bmatrix}\Delta\ddot{x} & \Delta\ddot{y} & \Delta\ddot{z}\end{bmatrix}^T = \mathbf{H}\begin{bmatrix}\nu_r & \delta_{r_t}\end{bmatrix}^T \quad (12)$$
$$\text{where } \nu_r = \begin{bmatrix}\Delta\theta & \Delta\phi & \Delta T\end{bmatrix}^T$$

The XINCA controller's actuator effectiveness highly depends on the current state of the vehicle, and needs to be recalculated at every time step. At low speeds aerodynamics do not play a great role yet, so it could be calculated as follows:

$$\mathbf{H} = \begin{array}{c} \\ \Delta\theta \quad\quad \Delta\phi \quad\quad \Delta T \quad\quad \delta_{r_t} \\ \begin{bmatrix} c\theta c\phi T & -s\theta s\phi T & s\theta c\phi & c\theta \\ 0 & -c\phi T & -s\phi & 0 \\ -s\theta c\phi T & -c\theta s\phi T & c\theta c\phi & -s\theta \end{bmatrix} \end{array} \begin{array}{c} \Delta\ddot{x} \\ \Delta\ddot{y} \\ \Delta\ddot{z} \end{array} \quad (13)$$

where s and c represent the sine and cosine functions respectively, and T represents the vertical specific force vector, which is estimated by taking the quadplane's vertical body acceleration and subtracting the gravitational acceleration:

$$T = \ddot{z} - g \quad (14)$$

When flying at higher velocities however, the quadplane will start to behave more like a fixed-wing aircraft. The incremental nature of the controller will automatically decrease the vertical thrust as the wings start to induce lift in order to maintain its vertical reference acceleration. A more drastic change in the quadplane's dynamics is the effect of a change in pitch, which starts to cause vertical acceleration. In order to include and utilise these changing dynamics, one term is added to the actuator effectiveness matrix, such that its final form is as follows:

$$\mathbf{H} = \begin{array}{c} \Delta\theta \quad\quad \Delta\phi \quad\quad \Delta T \quad\quad \delta_{r_t} \\ \begin{bmatrix} c\theta c\phi T & -s\theta s\phi T & s\theta c\phi & c\theta \\ 0 & -c\phi T & -s\phi & 0 \\ c\phi\left(\dfrac{C_{L_\alpha}\rho u^2 S}{2m} - s\theta T\right) & -c\theta s\phi T & c\theta c\phi & -s\theta \end{bmatrix} \end{array} \begin{array}{c} \Delta\ddot{x} \\ \Delta\ddot{y} \\ \Delta\ddot{z} \end{array} \quad (15)$$

where C_{L_α} is the change in lift per change in angle of attack, ρ is the air density, u again ideally is the true airspeed, S is the quadplane's wing surface area, and m is the platform's mass. Note that this matrix effectiveness is only a simplified estimation with which the XINCA controller should be able to appropriately control the quadplane. Looking at Equations 2a and 2b again: the XINCA optimisation parameters are chosen as follows:

$$\mathbf{W}_\tau = diag\begin{bmatrix}10 & 10 & 1\end{bmatrix}$$
$$\mathbf{W}_\delta = diag\begin{bmatrix}10 & 10 & 100 & 1\end{bmatrix}$$
$$\gamma = 10000$$
$$\delta_p = diag\begin{bmatrix}0 & 0 & 0 & 0\end{bmatrix}$$

$\mathbf{W}_\tau$ prioritises pitch and roll over thrust demands, since unstable flight might be more dangerous than a controlled descent. $\mathbf{W}_\delta$ penalises the use of pitch and roll and especially thrust commands with respect to using the tail rotor, and γ once again prioritises achieving the control demand over minimising the control effort. The maximum pitch and roll angles are set to 10°, the vertical thrust limits to -9.0 and 9.0 ms^{-2}, and the tail rotor's limits to 0 and 9600 PPRZ units. The actuator rate limits are again discarded.

To prevent the tail rotor hitting the ground, it is completely shut off for altitudes below 0.5 m with its effectiveness set to zero.

7 Flight Simulations

In order to prove XINCA's performance before taking flight, several simulations are performed. These simulations are executed within the Papparazzi UAV software in order to fully assess the performance of the actual code that will also fly on board of the quadplane. These simulation should mainly confirm three hypotheses:

- The INCA controller chooses suitable actuators in order to achieve stable flight
- The INCA controller chooses appropriate actuators in case of actuator saturation in order to maintain stable flight
- The XINCA controller chooses suitable actuators for forward flight

Ideally, the fourth hypothesis to be confirmed states that the XINCA controller controls vertical acceleration with thrust increments during vertical flight, and with pitch increments during horizontal flight. This research however does not make use of an appropriate simulation model including the dynamics of the quadplane's control surfaces, leaving this hypothesis open for future research. For the same reason, the control surfaces are excluded from the INCA optimisation during the simulations.

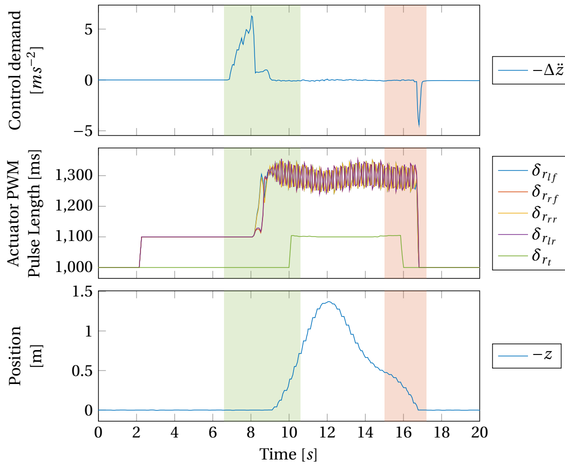


Figure 6: Simulation of a vertical quadplane takeoff and landing using XINCA and INCA with five actuators
(= takeoff, = landing)

Figure 6 shows a simple simulation of a quadplane taking off and landing right after. Indicated in the image are the takeoff (green) and landing (red) phases. The top plot shows the control demand the INCA controller aims to achieve. The middle plot shows the resulting actuator commands the INCA optimisation allocates, expressed in the PWM pulse length of a 400 Hz signal. The bottom plot shows the height profile of the flight. The simulation proves that INCA and XINCA can control a UAV in a stable manner. Note that the upwards facing rotors are rotating at their idle input level of 1100 ms before takeoff. As mentioned before, the tail rotor is only active for altitudes above 0.5 m.

In order to asses how well the INCA controller handles actuator saturation, a second simulation is performed with an artificial upper actuator limit slightly higher the nominal throttle level needed for hovering. The result can be seen in Figure 7. The figure shows that saturation occurs during takeoff, and that the low saturation limit also allows less fluctuation in the actuator commands. The UAV does have some trouble taking off, resulting in a slightly slower takeoff than in the simulation shown in Figure 6. The INCA controller however effortlessly achieves stable flight, since its pitch and roll commands are prioritised above its thrust and especially yaw commands.

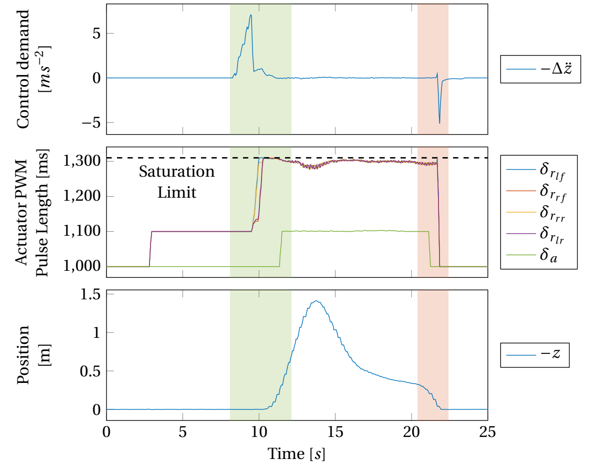


Figure 7: Simulation of a vertical quadplane takeoff and landing with actuator saturation occurring at an actuator PWM pulse length of 1310 ms using XINCA and INCA with five actuators
(= takeoff, = landing)

The third simulation aims to prove the applicability of the XINCA controller. In this situation, the UAV takes off after which it moves forwards, then twice as far backwards, and then back to its initial position where it lands. Figure 8 shows the results of this simulation, now with the control demand and position shown in the x-direction. This time the green and red areas show where the tail rotor is being activated by the XINCA controller for acceleration and braking respectively. In the middle plot it becomes clear that this happens exactly when expected. The tail ro-

tor is first activated to accelerate forward. The UAV then uses pitch increments to brake and accelerate backwards, after which it activates the tail rotor again twice in order to brake and move forward again. Finally, it slows down using pitch increments and lands.

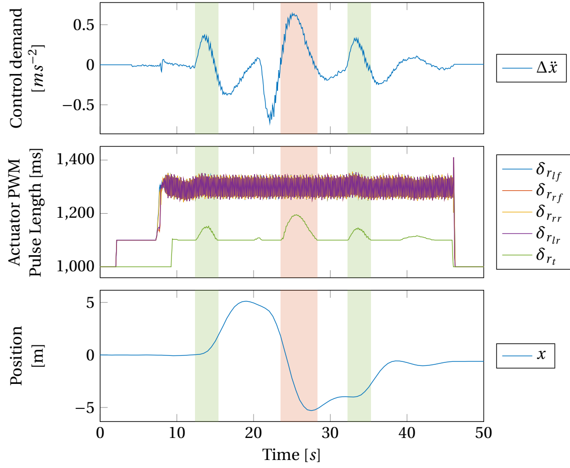


Figure 8: Simulation of forwards and backwards quadplane flight using XINCA with both pitch increments and tail rotor inputs and INCA with five actuators
(■ = forward acceleration by tail rotor, ■ = tail rotor braking)

Two last simulations are performed to clearly illustrate the benefits of using XINCA over conventional outer loop control methods. Both simulate forward flight of the quadplane, equipped with an outer loop INDI controller [33, 34] or the novel XINCA controller respectively. Using the main wing's aerodynamic properties in combination with the quadplane's pitch angle and forward velocity, an estimation is made of the hypothetical wing-induced lift force. The results are shown in Figure 10. This figure shows the actuator commands, pitch angle and lift force for both simulations. The most evident difference can be seen in the pitch angles. Where the INDI controller aggressively pitches forward to achieve forward acceleration, the XINCA controller proves to be able to minimise this negative pitch by using its tail rotor. This difference is reflected in the lift force estimations, where the XINCA controller manages to completely avoid negative lift caused by pitching forward. The INDI controller does inflict some negative lift, albeit of small magnitude. Note that this downforce might become more significant at larger velocities. The results of these last simulations are shown in Figure 11 as well, which plots flight profiles of both simulations and indicates pitch angles and lift forces. Again, it is clearly visible that the XINCA controller manages to minimise forward pitch and therefore negative lift, in contrast to the INDI controller.

8 Flight Experiments

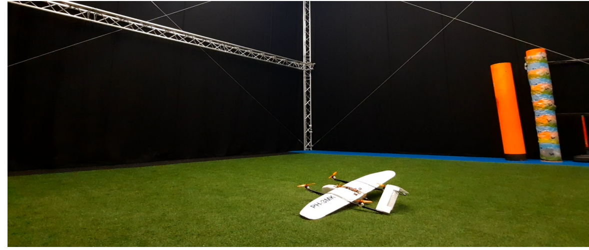


Figure 9: The TU Delft Quadplane in the Cyberzoo for light tests

The last step to be taken is to prove the airworthiness of both INCA and XINCA. This is done by equipping the TU Delft Quadplane with both controllers, and performing test flights in a controlled environment. The tests take place in a TU Delft facility called the *Cyberzoo*, which is a contained space equipped with

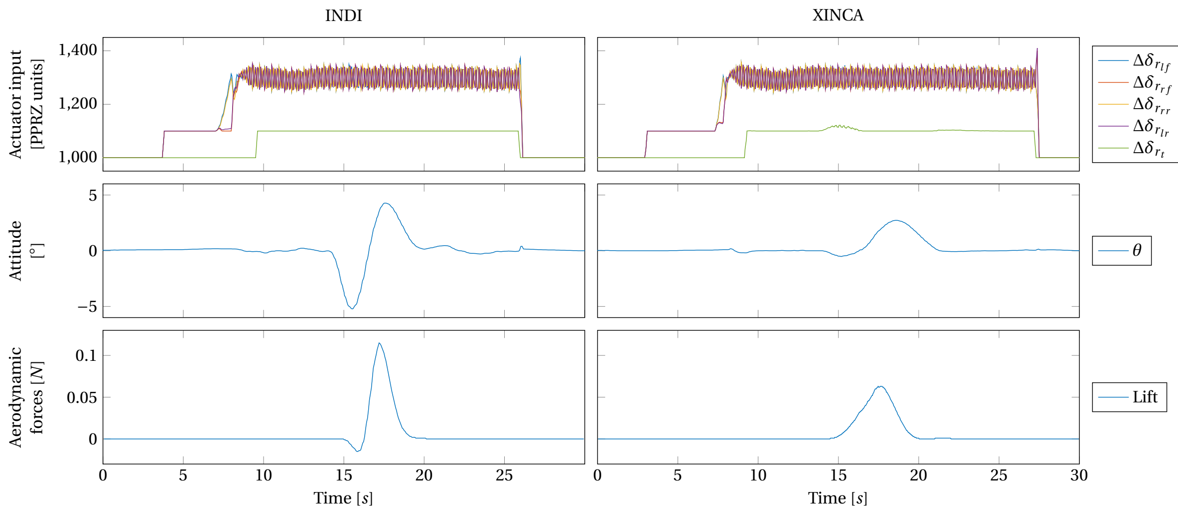


Figure 10: Flight data comparison of forward flight simulation with INDI and XINCA

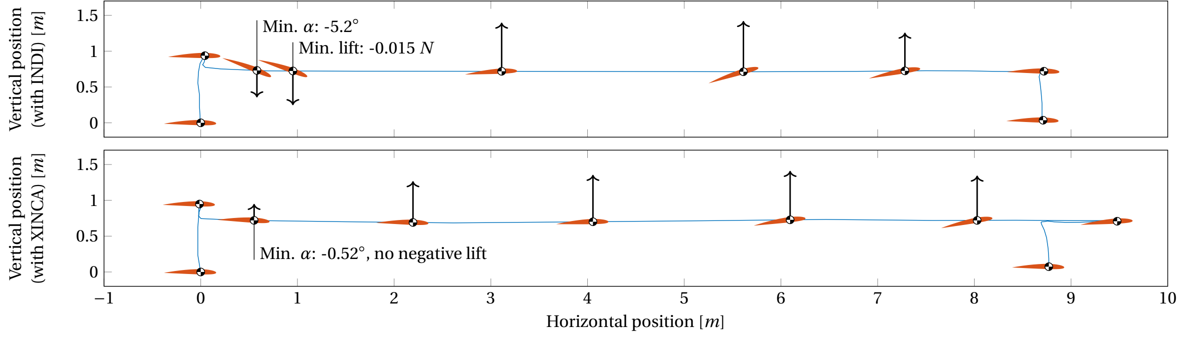


Figure 11: Flight profile comparison of forward flight simulation with INDI and XINCA showing wing-induced lift estimations Note that illustrated angles of attack are magnified and force vectors are scaled using an arctangent function for readability

an optical position tracking system for precise vehicle positioning. Figure 9 shows an image of the TU Delft Quadplane in the Cyberzoo.

During initial attempts to fly the Quadplane with both the INCA and XINCA controller, the vehicle often failed to respond to operator inputs, regularly resulting in crashes. This problem is most probably attributable to a lack of computing power of the quadplane's *Lisa/MX autopilot* board, which uses a 32-bit STM32-F4 CPU running at 266 MHz. The first measure to to reduce the computational cost of the controllers is to run the optimisations of both the inner and outer loops only once every second iteration of the autopilot, which runs at a cycle frequency of 512 Hz. An existing system monitoring module in Paparazzi has been used to estimate the autopilot's CPU loads with different configurations using this reduced optimisation frequency. These configurations include a combination of the INCA controller with a lower cost outer loop controller, a combination of a lower cost inner loop quadcopter controller with the XINCA controller, and a combination of both the INCA and XINCA controllers. For configurations using the INCA controller, the amount of INCA actuators is varied to determine its effect on computational cost. The results of these measurements can be seen in Table 1. Note that these measurements are taken on the quadplane itself, yet without taking off or flying.

	Inner loop:	INCA	Other	INCA
	Outer loop:	Other	XINCA	XINCA
INCA Actuators	4	38%	32%	48%
	5	46%		54%
	6	54%		62%
	7	62%		71%
	8	74%		83%

Table 1: CPU load estimations for different inner and outer loop controllers with different numbers of INCA actuators

Because of the Active Set Method, the numbers clearly show a quasi-linear correlation between the number of actuators and the CPU load, and that the configuration with both INCA and XINCA does indeed demand a lot of the autopilot's computing power. The fact that the maximum recorded CPU load is still well below 100% can be explained by the fact that the optimisation schemes only run once every two cycles, resulting in an average load under 100%. The actual load during one optimisation cycle might however require significantly more computing power, resulting in unpredictable behaviour of the quadplane. Especially some time-critical processes need to be re-evaluated in order to perform well under high CPU load. Ideally, the quadplane's autopilot board is to be replaced by one with sufficient computing power. For this research however, flight tests will be performed with either both INCA and XINCA without any control surfaces, or INCA with all inner loop actuators and a low cost outer loop controller.

Like the first simulation discussed in Section 7, the first test flight aims to confirm that the INCA controller chooses suitable actuators during flight. All inner loop actuators are included in the optimisation, so the low cost outer loop controller is used for this test. Since the Cyberzoo's confined space only allows for low velocity testing however, the controller is not expected to allocate a significant amount of control to the control surfaces. The results in Figure 12 confirm this. They show varying inputs for the quadplane's upwards facing rotors, due to a slight asymmetrical configuration, but prove to successfully ensure a stable takeoff and landing. The control surfaces only show minor variations as expected, except the landing phase. As soon as the Quadplane touches down, it becomes more or less static, resulting in an unachieved control demands. This causes the rotors to saturate at their minimum values, after which the control surfaces are saturated as well in a maximum effort to satisfy the control demand.

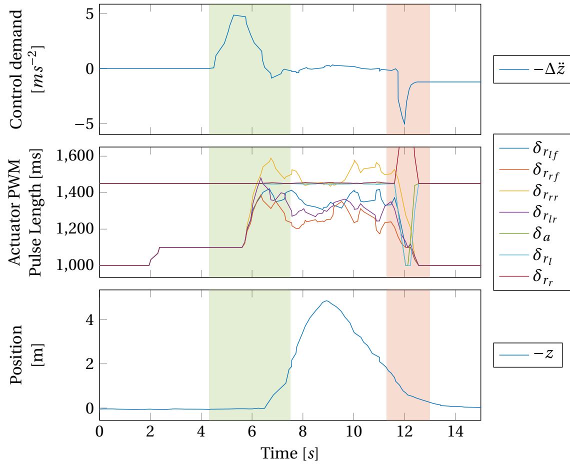


Figure 12: Stable quadplane takeoff and landing using INCA with seven actuators (■ = takeoff, ■ = landing)

The difference in actuator inputs between different rotors seen in the first flight can be exploited in the second, where INCA's resilience against actuator saturation is being put to the test. The saturation level is chosen in such a way that one actuator especially saturates, in this case $\delta_{r_{lr}}$. Like with its corresponding simulation, Figure 13 shows that INCA prioritises its pitch and roll commands above its thrust and especially yaw commands, resulting in slower but stable takeoff. Saturating actuators however result in the INCA optimisation having to perform more iterations before it reaches its optimum, since the Active Set Method has to explore the edges of the actuator input space in multiple steps. This eventually results in higher computational cost. This test is therefore performed with the INCA controller using only four actuators and a low cost outer loop controller.

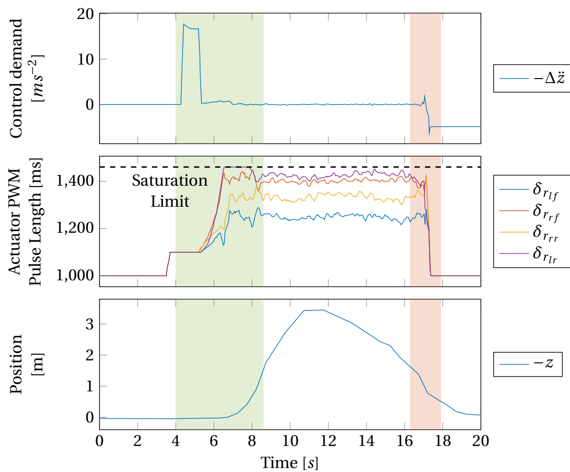


Figure 13: Stable quadplane takeoff and landing with actuator saturation occurring at an actuator PWM pulse length of 1460 ms using INCA with five actuators (■ = takeoff, ■ = landing)

The final performed flight is the one where the novel XINCA module is being tested. For this flight, the quadplane is controlled by both the INCA and XINCA controllers that together allocate control to a total of five rotors. The flight itself consists of a takeoff, forward flight, backwards flight and landing. Figure 14 shows that the quadplane effortlessly manages to perform this longitudinal manoeuvre. Peaks in the tail rotor command show that this actuator is indeed used for both forward acceleration and backwards braking as expected.

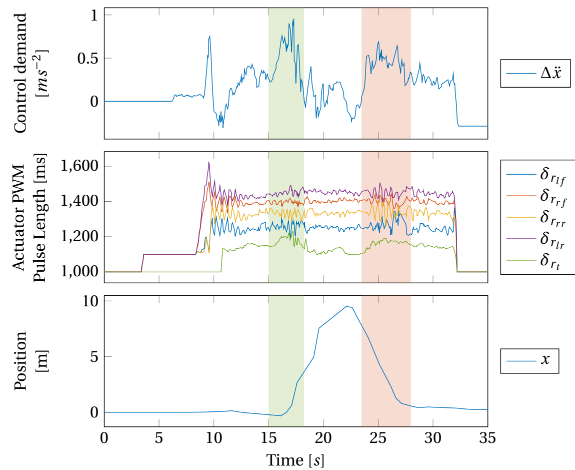


Figure 14: Forwards and backwards quadplane flight using XINCA with both pitch increments and tail rotor inputs and INCA with five actuators (■ = forward acceleration by tail rotor, ■ = tail rotor braking)

9 Conclusions and Recommendations

As mentioned earlier, this research aims to prove three hypotheses:

- The INCA controller chooses suitable actuators in order to achieve stable flight
- The INCA controller chooses appropriate actuators in case of actuator saturation in order to maintain stable flight
- The XINCA controller chooses suitable actuators for forward flight

During both the simulations and the actual test flights all three have been confirmed, albeit with some side notes. First of all, the existing INCA controller performs well for both quadcopter and quadplane configurations. It proves to not require very detailed models of its controlled vehicle, and the Active Set Method makes it suitable for real-time optimisation at high frequencies. Recalculation of the actuator's effectiveness at every time step results in a

high automated adaptability to changing states and conditions to ensure efficient flight control, using the most suitable and efficient actuators available. When optimising commands for too many actuators however, this INCA controller is not efficient enough to be used on the TU Delft Quadplane in its current hardware configuration. Allocating control to seven actuators while using a low cost outer loop controller is at the edge of its computational capacity. Future research on this specific platform therefore requires hardware upgrades to achieve more computing power.

INCA also proves to handle actuator saturation well. Prioritising certain control demands over others successfully ensures stable flight when saturation occurs. This also makes platforms like the quadplane more resilient towards decreased actuator effectiveness as a result of damages, or to windy conditions that could make it impossible for certain control demands to be achieved.

Finally, the novel XINCA controller proves capable to perform an optimisation in the outer control loop, similar to the one in the INCA inner control loop. It shows it can use a combination of increments in attitude angles and actual actuators to achieve an outer loop control demand containing increments in the three linear accelerations. This method eliminates the inefficient use of separated flight modes, resulting in better performance of hybrid vehicles.

Future research on the application of INCA on hybrid vehicles like the quadplane and the application of XINCA in general should focus on their performance during level flight, as this has not been sufficiently addressed during this research. Outdoor flights should serve two main research objectives. One objective would be to assess how the quadplane allocates more control to its control surfaces, as soon as it has an amount of forward airspeed making them more effective. The other objective focuses on XINCA, assessing its capabilities to adapt to the different dynamics of a hovering quadplane and one in forward flight. The hypothesis to be proved is that reducing the required input of either a linear outer loop controller or human operator to only linear to be achieved reference accelerations is indeed beneficial, and that XINCA performs well durng different flight phases.

INCA has once again been proven promising in the field of UAV control, especially for over-actuated vehicles. Its novel extension called XINCA has shown the potential to even further increase the capabilities of hybrid UAVs. With proper research on both controller designs they might prove to be suitable and applicable to not only UAVs but innovative manned vehicles as well. This could potentially contribute significantly to a safer, more efficient and therefore greener future of human aerial transportation.

Bibliography

[1] Elisabeth Bumiller and Thom Shanker. War evolves with drones, some tiny as bugs. *The New York Times*, Jun 2019. URL `https://www.nytimes.com/2011/06/20/world/20drones.html`.

[2] Amarjot Singh, Devendra Patil, and SN Omkar. Eye in the sky: Real-time drone surveillance system (dss) for violent individuals identification using scatternet hybrid deep learning network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[3] A. Momont. Drones for good. Master's thesis, Delft University of Technology, 2014.

[4] Lilium, 2020. URL `https://lilium.com/`.

[5] U.M. Rao Mogili and B.B.V.L. Deepak. Review on application of drone systems in precision agriculture. *Procedia Computer Science*, 133:502–509, 2018. doi: 10.1016/j.procs.2018.07.063.

[6] Elisabeth Bumiller and Thom Shanker. War evolves with drones, some tiny as bugs. *The New York Times*, Jun 2019. URL `https://www.nytimes.com/2011/06/20/world/20drones.html`.

[7] Sasanka Madawalagama, Niluka Munasinghe, S Dampegama, and L Samarakoon. Low cost aerial mapping with consumer-grade drones. 10 2016.

[8] Zhang Daibing, Wang Xun, and Kong Weiwei. Autonomous control of running takeoff and landing for a fixed-wing unmanned aerial vehicle. *2012 12th International Conference on Control Automation Robotics & Vision (ICARCV)*, 2012. doi: 10.1109/icarcv.2012.6485292.

[9] Marco Palermo and Roelof Vos. Experimental aerodynamic analysis of a 4.6%-scale flying-v subsonic transport. *AIAA Scitech 2020 Forum*, May 2020. doi: 10.2514/6.2020-2228.

[10] D. Jin Lee, Byoung-Mun Min, Min-Jea Tahk, Hyochoong Bang, and D.h Shim. Autonomous flight control system design for a blended wing body.

2008 International Conference on Control, Automation and Systems, 2008. doi: 10.1109/iccas.2008.4694548.

[11] Bai Zhiqiang, Liu Peizhi, Wang Jinhua, and Hu Xiongwen. Simulation system design of a uav helicopter. *2011 International Conference on Electric Information and Control Engineering*, 2011. doi: 10.1109/iceice.2011.5778108.

[12] Teppo Luukkonen. Modelling and control of quadcopter, Aug 2011.

[13] E.J.J. Smeur, D.C. Höppener, and C. De Wagter. Prioritized control allocation for quadrotors subject to saturation. *International Micro Air Vehicle Conference and Flight Competition (IMAV)*, 2017.

[14] Jacob Apkarian. Attitude control of pitch-decoupled vtol fixed wing tiltrotor. *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2018. doi: 10.1109/icuas.2018.8453473.

[15] Ryuta Takeuchi, Keigo Watanabe, and Isaku Nagai. Development and control of tilt-wings for a tilt-type quadrotor. *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*, 2017. doi: 10.1109/icma.2017.8015868.

[16] Christophe De Wagter and Ewoud J.J Smeur. Control of a hybrid helicopter with wings. *International Journal of Micro Air Vehicles*, 9(3):209–217, Nov 2017. doi: 10.1177/1756829317702674.

[17] Matthew E. Argyle, Jason M. Beach, Randal W. Beard, Timothy W. Mclain, and Stephen Morris. Quaternion based attitude error for a tailsitter in hover flight. *2014 American Control Conference*, 2014. doi: 10.1109/acc.2014.6859324.

[18] Danial Sufiyan Bin Shaiful, Luke Thura Soe Win, Jun En Low, Shane Kyi Hla Win, Gim Song Soh, and Shaohui Foong. Optimized transition path of a transformable hovering rotorcraft (thor). *2018 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2018. doi: 10.1109/aim.2018.8452703.

[19] Janith Kalpa Gunarathna and Rohan Munasinghe. Development of a quad-rotor fixed-wing hybrid unmanned aerial vehicle. *2018 Moratuwa Engineering Research Conference (MERCon)*, 2018. doi: 10.1109/mercon.2018.8421941.

[20] Jian Zhang, Zhiming Guo, and Liaoni Wu. Research on control scheme of vertical take-off and landing fixed-wing uav. *2017 2nd Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*, 2017. doi: 10.1109/acirs.2017.7986093.

[21] David Orbea, Jessica Moposita, Wilbert G. Aguilar, Manolo Paredes, Rolando P. Reyes, and Luis Montoya. Vertical take off and landing with fixed rotor. *2017 CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, 2017. doi: 10.1109/chilecon.2017.8229691.

[22] Ma Tielin, Yang Chuanguang, Gan Wenbiao, Xue Zihan, Zhang Qinling, and Zhang Xiaoou. Analysis of technical characteristics of fixed-wing vtol uav. *2017 IEEE International Conference on Unmanned Systems (ICUS)*, 2017. doi: 10.1109/icus.2017.8278357.

[23] Gerardo Flores and R. Lozano. Lyapunov-based controller using singular perturbation theory: An application on a mini-uav. *2013 American Control Conference*, 2013. doi: 10.1109/acc.2013.6580063.

[24] A.R.J. Stolk. Minimum drag control allocation for the innovative control effector aircraft. Master's thesis, Delft University of Technology, 2017.

[25] Ewoud J.J. Smeur, Qiping Chu, and Guido C.H.E. De Croon. Adaptive incremental nonlinear dynamic inversion for attitude control of micro air vehicles. 2015. doi: https://doi.org/10.2514/1.G001490.

[26] Joseph Horn. Non-linear dynamic inversion control design for rotorcraft. *Aerospace*, 6(3):38, 2019. doi: 10.3390/aerospace6030038.

[27] D.C. Höppener. Actuator saturation handling using weighted optimal control allocation applied to an indi controlled quadcopter. Master's thesis, Delft University of Technology, 2017.

[28] Ola Härkegård. Dynamic control allocation using constrained quadratic programming. *AIAA Guidance, Navigation, and Control Conference and Exhibit*, May 2002. doi: 10.2514/6.2002-4761.

[29] Henri P. Gavin and Jeffrey T. Scruggs. Constrained optimization using lagrange multipliers. *CEE 201L. Uncertainty, Design, and Optimization*, 2020.

[30] Tor A. Johansen and Thor I. Fossen. Control allocation—a survey. *Automatica*, 49(5):1087–1103, 2013. doi: 10.1016/j.automatica.2013.01.035.

[31] O. Harkegard. Efficient active set algorithms for solving constrained least squares problems in aircraft control allocation. *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, 2002. doi: 10.1109/cdc.2002.1184694.

[32] Pascal Brisset, Antoine Drouin, Michel Gorraz, Pierre-Selim Huard, and Jeremy Tyler. The paparazzi solution *. 10 2006.

[33] Ewoud J.J. Smeur, Qiping Chu, and Guido C.H.E. De Croon. Cascaded incremental nonlinear dynamic inversion control for mav disturbance rejection. 2018. doi: https://doi.org/10.1016/j.conengprac.2018.01.003.

[34] Ewoud J.J. Smeur, Qiping Chu, and Guido C.H.E. De Croon. Gust disturbance alleviation with incremental nonlinear dynamic inversion. 2016. doi: https://doi.org/10.1109/IROS.2016.7759827.

II

Part II: Preliminary Research Report

as graded for the AE4020 Literature Study

1
Abstract

The report presents a literature research that was performed as part of a TU Delft Master Thesis at the faculty of Aerospace engineering. The objective of the thesis work is to assess the suitability of Incremental Non-linear Control Allocation or INCA on the over-actuated TU Delft quadplane. This literature research focuses on gaining relevant knowledge about control allocation, quadplane controllers in general, and INCA specifically. It addresses suitable optimisation techniques that can be used in control allocation, and proper definitions of control allocation optimisation problems. Also, a proposal for further research is made.

2
Introduction

Over the last decade, Unmanned Aerial Vehicles (UAVs) have been gaining popularity at an incredible rate. New applications are found almost every day, ranging from consumer to commercial and even military purposes. Some examples of applications are reconnaissance, attack, film making, law enforcement, research, surveillance, agriculture, construction and many more. While UAV technology advances, the requirements for such platforms become more and more demanding. End users expect their UAVs to fly longer and over larger distances, while flying ever smoother and more stable. They often have to be versatile and be able to operate under harsh flight conditions. These and many other application-specific requirements push researchers and developers to come up with innovative solutions, in terms of both airframe design and control strategies. This has led to many different types of UAVs, each with its own advantages and disadvantages.

2.1. UAV Classification

An overview of some of the most common types of UAVs is given in Figure 2.1. One can see that the majority of UAVs consist of rotorcraft and fixed wing UAVs.

Figure 2.1: UAV classification

Rotorcraft make use of one or more rotors to generate lift. This provides them with VTOL capabilities, allowing them to take off and land virtually anywhere. The most common types of rotorcraft would be the conventional helicopter as researched by [1], and the multicopter (often quadcopter) as researched by [3, 4]. Possibilities of less conventional types of rotorcraft have also been explored, often in an attempt to satisfy very mission specific requirements. Results are for instance the monocopter designed by Lembono et al. [2], which mimics a single seed or *samara* from maple trees. This example proves to be mechanically simple, but unsurprisingly challenging to control. Other less conventional types with their own advantages and disadvantages are ducted fan rotorcraft [5], autogyros, also known as gyrocopter or gyroplanes [6] and cyclogyros [7].

Fixed wing UAVs resemble the conventional airplane design, using horizontal wings for lift generation and control surfaces for attitude control. They often need an airstrip for takeoff and landing, but their wing-

induced lift instead of active vertical thrust results in better endurance performance, i.e. longer flight ranges and flight time. They often come in the conventional aircraft form with a fuselage, wings and stabilisers [18], but sometimes also in the more progressive form of a blended wing aircraft [19, 20]. Occasionally, rotor-less VTOL capabilities are incorporated into the design by means of thrust vectoring [21], where the thrust vector of a power unit can be deflected towards its desired direction.

While completely distinct UAV platforms exist like flapping wing UAVs or ornithopers [22], hot air or helium balloons [23] and zeppelins [24], the most relevant UAV platforms for this research are those hybrid between rotorcraft and fixed wing UAVs. It appears that hybrid UAVs, like the TU Delft quadplane have gained quite some popularity in recent years. They aim to combine the best of two worlds, namely the VTOL capabilities of rotorcraft, and the endurance of fixed wing aircraft. This can for instance be achieved by tilting the aircraft's rotors or even its complete wing upwards, and back to their horizontal position during flight [8, 9]. This very effective method is able to reduce the amount of rotors needed and wing-induced drag during vertical climb, but at the cost of additional mechanically complexity, and therefore additional mass. Letting a winged aircraft take off vertically can otherwise be achieved by letting the whole UAV take off from a vertical or *tail sitting* orientation, and rotating the whole platform during climb towards a horizontal orientation [10, 11]. This provides the potential for mechanical simplicity, albeit that the UAV and its payload need to withstand a vertical attitude. Another solution is a range of transformable UAV, which completely change its configuration during flight in order to switch between vertical and horizontal flight. One remarkable example is the one designed by Shaiful et al. [12], where the UAV takes off as a two-bladed rotor with active propulsion on each of the blades. When transitioning to horizontal flight, one of the blades rotate around its own axis to form a fixed wing together with the other blade, and provide propulsion in one longitudinal direction. The last type of UAV from the overview in Figure 2.1 to discuss is the one being used in this research, the quadplane. This platform will be discussed in the next section.

The amount of different hybrid UAV designs is remarkable. Every type has its own benefits, but also its drawbacks. Researcher seem to be struggling to find one optimal platform. The reason is that harmonising the dynamics of both vertical and horizontal flight is quite challenging. The dynamics of a rotorcraft is per definition quite different from those of fixed wing UAVs. Their conventional controllers are governed by control laws that conflict when used together during a transition. In order to provide the yet non-existing solutions to these problems, it is therefore important that research keeps being performed on novel controller types for hybrid air frames. This way, this type of UAV will be able to fly longer, further and smoother. Occasionally, companies scale up aircraft platforms that so far have only been unmanned, in order for them to carry passengers. Good examples of such companies are Lilium [25] and Ehang [26]. Their products might eventually contribute towards an all-electric future of air transport, a reduction of car traffic and a more sustainable environment. These kinds of platforms only exist because of immense amounts of UAV research preceding them. Needless to say, ongoing research on quadplanes and UAVs in general remains not only of commercial or scientific, but also of public interest.

2.2. The Quadplane

The quadplane masters both versatility and endurance without too much mechanical complexity. An example is the one developed for research purposes by the Delft University of Technology's MAVLab, henceforth called the TU Delft quadplane. Its versatility results from the combination of a quadrotor and a fixed-wing UAV, providing it with both Vertical Takeoff and Landing (VTOL) capabilities and efficient and enduring level flight performance. Its simplicity is because of the fact that it is based on conventional UAV concepts, and does not need to have mechanically complex parts like tilting wings or rotors. The latter also positively influences the total weight of the platform. It does however have some other additional mass, caused by redundant actuators, proving the struggle to find an absolutely optimal hybrid UAV airframe.

The design of the TU Delft quadplane specifically is described by Aman [27]. The upward facing rotors are attached to two longitudinal rods, reducing the amount of additional drag induced by these rotors. It is controlled by nine actuators in total, as schematically illustrated in Figure 2.2. Four upwards facing rotors are typically used for vertical takeoff, landing and hovering. A tail rotor, two ailerons and two ruddervators are used for level flight. These sets will respectively be referred to as the quadcopter actuator set and the fixed

wing actuator set.

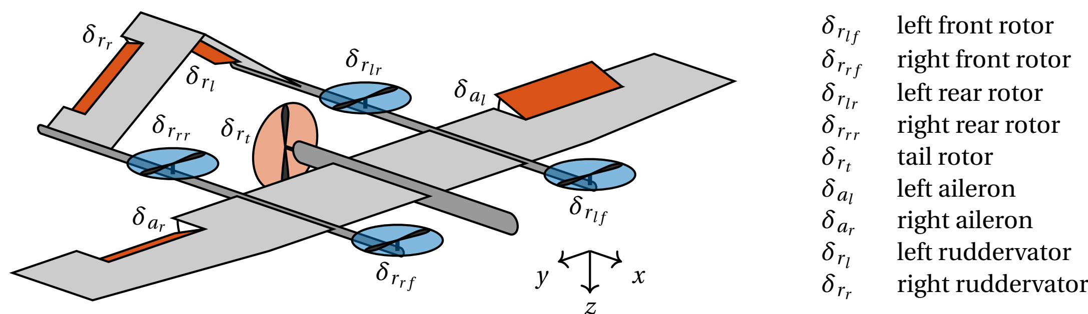


Figure 2.2: Overview of the quadplane's nine actuators
(= quadcopter actuator set, = fixed wing actuator set)

Because the quadplane has more actuators than it actually needs to control its attitude, it is called *over-actuated*. An aircraft's controller decides what actuators need to be activated and to what extent, in order to achieve the desired moments or *control demand* acting on the system. For aircraft that are not over-actuated this is straight-forward, as each of the degrees of freedom is typically being accounted for by only a single actuator or set of actuators. One example is a conventional aircraft, which has elevators to control pitch, ailerons to control roll and a rudder to control yaw. Another example is a quadcopter, which uses differential thrust between front and aft rotors to control pitch, differential thrust between left and right rotors to control roll and differential thrust between clockwise and counter-clockwise rotating rotors to control yaw. Controlling an over-actuated UAV however, is less straight-forward. A quadplane controller, for instance, constantly needs to choose whether to use its quadcopter actuator set, its fixed wing actuator set or a combination of the two to control its attitude. This can for example be based on the actuators' effectiveness, which for control surfaces heavily depends on forward airspeed.

Problems might also occur when the actuators have a higher control demand than they can satisfy. This is called the *saturation problem*. This problem can potentially occur in every controlled system, but a quadplane is especially prone to it. An exemplary situation for both quadplanes in hover (only using their quadcopter actuator set) and quadcopters could be when their controller has to satisfy multiple control demands in pitch, roll, yaw and/or thrust. The active actuators are all involved in satisfying each of these demands. Satisfying all of them could be more than certain actuators can handle, resulting in saturation of one or more actuators and at least some control demands not being achieved. Another example that is specific to quadplanes, is saturation that might occur when controlling the platform's attitude in windy conditions while hovering. Because of the large wing and control surfaces of the platform, large forces and moments caused by wind can act on the system. Especially yawing forces caused by wind on the tail sector of the quadplane are difficult to counter-act, as the quadplane's quadcopter actuator set is least effective in yaw. This could be compensated by taking bigger and more powerful rotors. This is however undesirable, since they add a lot of extra weight to the system and increase drag during horizontal flight, although they are mostly used for the relatively short take off and landing phases. A better solution would be to optimally use all actuators to achieve control demands a well as possible, with a controller that is able to prioritise certain control demands over others. It could potentially use quick attitude changes in order to gain control effectiveness in a desired direction, for instance: a quadplane could quickly pitch, roll and pitch back sequentially in order to help satisfy a control demand in yaw.

Choosing or combining multiple actuators to satisfy a certain control demand is called *control allocation*, and will be discussed in Section 3. Often, some sort of optimisation method needs to be applied in order to achieve optimal control allocation. Suitable methods for this purpose are described in Section 4. Further research and experiments to be performed during the remainder of the thesis project are described in Section 5, and this literature research is concluded in Section 6

3

Control Allocation

When performing research on existing quadplane solutions, one can find a modest amount of beautifully designed quadplanes. A first example is the one designed by Gunarathna and Munasinghe [13], who designed adequate controllers for their hybrid platform, which is a modified fixed wing UAV. This quadplane has two distinct controllers. One to control the its quadrotor actuator set during take-off, hovering and landing, and one to control its fixed wing actuator set during horizontal flight. During a transition phase, it shortly uses both controllers in order to gain speed and therefore wing-induced lift, or to slow down and compensate the decrease in lift with the quadcopter actuator set.

A similar example is the quadplane designed by Zhang et al. [14], who distinguishes four different flight modes: a multi-rotors mode, fixed-wing mode, and the front and back transition modes. This literature shows challenges induced by simultaneously using two controllers during a transition phase. For instance, heading control is quite different in the two main flight modes, potentially resulting in undesirable control behaviour of the quadplane during transitioning. Also, the fixed-wing mode can produce high pitching angles during transitioning, due to the lack of wing-induced lift. This could potentially result in the aircraft stalling and becoming unstable.

Slightly different is the platform designed by Orbea et al. [15], although it could still be qualified as a quadplane. It is however different than the quadplanes discussed so far, in the sense that it does not feature horizontal propulsion. The quadplane's fixed wing only provides additional lift during horizontal flight. Because of the forward pitching that quadrotor use to gain a forward acceleration, the wing is attached to the airframe at a high inclination angle. This way, a positive wing-induced lift force is assured. Although this UAV technically is a quadplane, it is not over-actuated and could be controller by a single attitude controller.

Tielin et al. [16] provide an overview of different UAVs with VTOL capabilities. In their analysis of UAVs that feature *seperate power plant behaviour for hover"* like a quadplane, they state that that these platforms have two flight control systems, like most of the platforms named above. Only during some transition phase, their actuators get combined to move from one mode to another. Constantly allocating control to all available actuators might improve overall effectiveness and efficiency. This topic will be further elaborated this section.

First, some background on controlling UAVs is required. A typical UAV controller consist of at least two control loops, as can be seen in Figure 3.1. The outer or position loop is used for guidance, and will aim to compensate any positional errors. It does so by providing a reference attitude and a direct lift or thrust command to the inner or attitude loop, which is used for stabilisation of the UAV. This loop controls the attitude of the UAV by directly controlling the UAV's actuators [28].

This research focuses on the inner loop controller, the one that controls the UAVs attitude. Different existing solutions will be presented in this section, starting with ones that are not able to deal with over-actuation, and ending with Incremental Non-linear Control Allocation or INCA, which does.

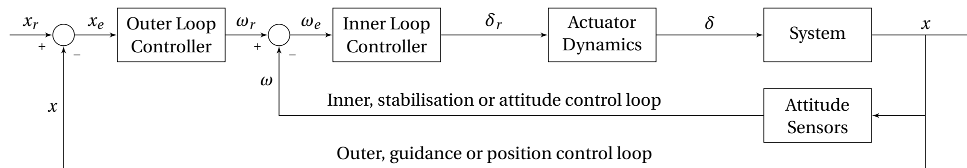


Figure 3.1: A schematic representation of a controller with inner and outer control loops
(x = state vector, ω = attitude vector, δ = control input vector)

3.1. The PID controller

Perhaps one of the most conventional controller methods is the Proportional-Integral-Derivative or PID controller as described by Araki [29]. This controller takes a state error as input, and defines the reference control input as a linear combination of the error itself, its integral and its derivative:

$$\delta_r = K_P \omega_e + K_I \int \omega_e + K_D \frac{d\omega_e}{dt} \tag{3.1}$$

in which ω_e could be an error in a certain attitude angle, and δ_r could be a corresponding reference control input. The PID controller could be implemented in a system as depicted in Figure 3.2. An advantage of a PID controller is that it somewhat anticipates on approaching a reference state, since it not only takes the state error into account, but also the rate at which this error decreases. Also, the PID controller can compensate any residual error since it takes the error integral into account as well. The main advantage however is that a PID controller does not need a detailed model of the controlled system, and that it is directly applicable to not just first order, but also second order systems.

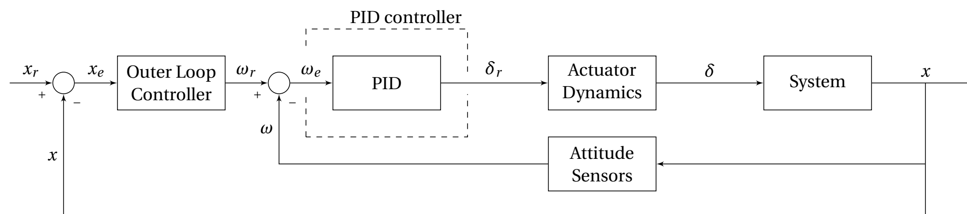


Figure 3.2: A schematic representation of a PID controller
(x = state vector, ω = attitude vector, δ = control input vector)

The PID controller only works when certain state errors correspond exclusively to certain control inputs. For instance, ailerons can be used for roll control, elevators for pitch control and a rudder for yaw control. A quadplane's alternative is to link certain upwards facing rotor thrust differentials to the attitude errors. A PID controller is however not able of making a smart choice between multiple available actuators. To use it in the attitude control loop of over-actuated UAVs, one would need to switch between different PID controllers, as done by Gunarathna and Munasinghe [13] and Zhang et al. [14]. A quadcopter controller could be used to taking off, landing and hovering, and a fixed wing controller could be used for horizontal flight. Only in a transition phase might these actuator be used simultaneously, for instance to gain forward velocity before switching from the quadcopter to the fixed wing controller.

Another disadvantage of the PID controller is that it does not use any knowledge about the system to be controlled. Controller gains are often tuned manually, in a trial-and-error kind of way. This is a tedious and time consuming process, often resulting in sub-optimal performance. PID controllers in general are also not capable of taking actuator constraints into account, possibly resulting in saturated actuators and insufficient control over the UAV in question. Lastly, PID controllers can only react to feedback of the system, while model-based controllers, like the ones described later, can also incorporate some sort of prediction or feed-forward.

3.2. The NDI controller

Often, a mathematical description is known of the system to be controlled. This model might give a controller designer a lot of information on how to best control its vehicle. Let for instance a system be described as a linear state-space system:

$$\nu = \mathbf{F}x + \mathbf{G}\tau \tag{3.2a}$$

$$\text{where } \tau = \mathbf{H}\delta \tag{3.2b}$$

In these equations, ν conventionally contains the system's angular accelerations, τ is the vector containing the the actuator moments acting on the system, and δ is the control input vector. State matrix $\mathbf{F}$, inertial matrix $\mathbf{G}$ and actuator effectiveness matrix $\mathbf{H}$ are state and time dependant system matrices. An NDI controller receives a so-called virtual input ν_r, and needs to output a control input vector that satisfies this demand. It does so by first determining a *control demand* τ_c by inverting Equation 3.2a, and substituting this control demand into Equation 3.2b, which is than in turn inverted:

$$\tau_c = \mathbf{G}^{-1}(\nu_r - \mathbf{F}x) \tag{3.3}$$

$$\delta = \mathbf{H}^{-1}\tau \tag{3.4}$$

A schematic representation of a system with an NDI controller is shown in Figure 3.3. The controller now has based a control input solely on a the virtual input ν_r, state vector x and the mathematical model of the system. An example of an NDI implementation in the one by Horn [30] on a quadcopter platform, and is used in many other UAV systems. It eliminates tedious tuning of controller gains as encountered in PID controllers. It does however require an accurate model of the system to be controlled. If certain dynamics of a system are poorly modelled, the system might not be controlled as desired, or even become unstable. This might also be the case when other moments than those taken into account in the model are acting on the system. Another disadvantage of this method, is that it needs invertible actuator dynamics, which is per definition not the case for over-actuated systems, where the number of actuators exceeds the number of degrees of freedom the system has. To work around this in an NDI controller, one could group multiple actuators to share one control input value, or disable certain actuators in certain flight modes. These methods would however be sub-optimal, since they are not able to combine certain actuators. Lastly, the NDI controller in itself has no way to take actuator constraints into account.

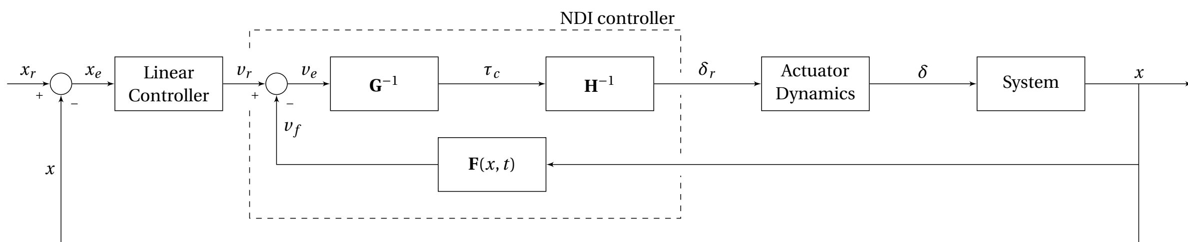


Figure 3.3: A schematic representation of an NDI controller
(x = state vector, ν = virtual input, τ = control demand, δ = control input vector)

3.3. The INDI controller

In order to reduce the model dependency an NDI controller has, one can choose to substitute the system's model dynamics with measurements, so the only models that are required are those of the actuators. Vehicles often have an Inertial Measurement Unit or IMU on board, which measures specific moments acting on the system. Especially now, the convenience of using a virtual control input ν_r becomes clear. The amount of angular accelerations that is yet to be achieved can be defined as the difference ν_e between this virtual control input ν_r and measured specific moments ν. Note that unlike an NDI controller, this ν_e denotes the required *error* in angular accelerations to be compensated by actuator *increments*, and not the total desired

angular accelerations to be compensated by absolute actuator positions. This is because the IMU sensors also measure the effects of the current actuator positions. The calculated control input vector is thus also incremental, and denoted as $\Delta\delta_r$. It is thereafter added to the current measured or estimated control input δ. This incremental variation of the NDI method is unsurprisingly called Incremental Non-linear Dynamic Inversion, or INDI. A diagram of a system with an INDI controller is shown in Figure 3.4.

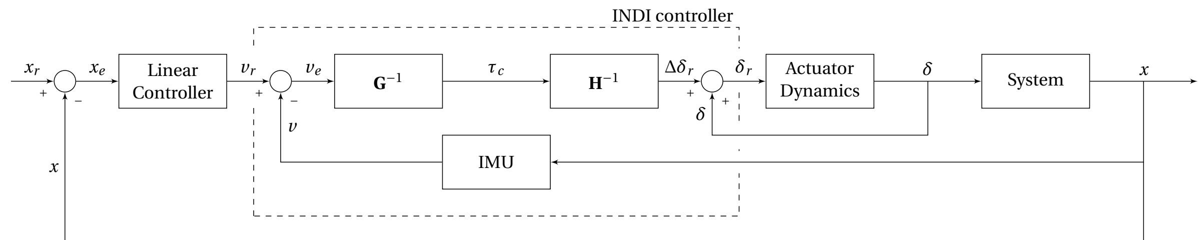


Figure 3.4: A schematic representation of an INDI controller
(x = state vector, v = virtual input, τ = control demand, δ = control input vector)

The main advantage of this method is that it captures unmodelled or unanticipated dynamics by measuring what states instead of trying to predict them. Not also does this mean that the model of the system's dynamics is not needed anymore, but also that the actuator model does not need to be very precise. The virtual input is constantly compared to actual measurements, making sure that inaccuracies in the model will eventually be compensated whenever possible. However, the inversion of the so-called actuator effectiveness matrix **H** still prohibits proper application of the controller on over-actuated system such as the quadplane, for the same reasons as an NDI controller. Höppener [31] shows an implementation of this method on a quadcopter. In his thesis, he addresses actuator saturation as well. He proposes the use of a Weighted Least Squares or WLS control allocation, which allows finding a optimal control input that respects actuator constraints. This extension to the INDI controller is called Incremental Non-linear Control Allocation or INDI, and is discussed in the next chapter.

3.4. The INCA controller

Incremental Non-linear Control Allocation, or INCA, is based on the INDI controller. The only difference is that the inversion of actuator effectiveness matrix **H** is replaced by a control allocation optimisation scheme, as can be seen in Figure 3.5. This is because matrix **H** is per definition singular for over-actuated systems. This is not the case for inertial matrix **G**, which typically is an invertible 3×3 matrix that correlates moments acting on the system to angular accelerations.

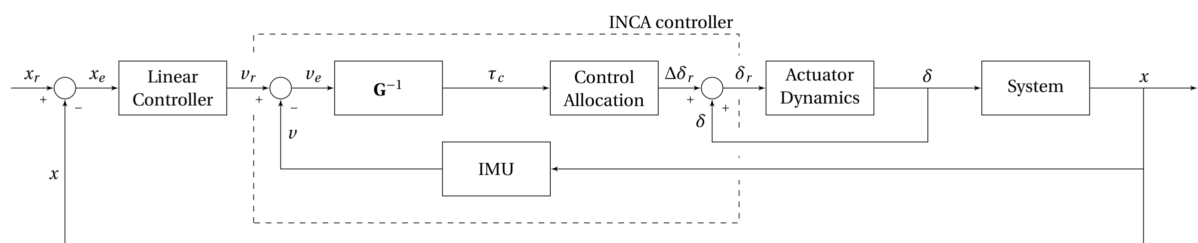


Figure 3.5: A schematic representation of an INCA controller
(x = state vector, v = virtual input, τ = control demand, δ = control input vector)

What this control allocation optimisation actually encompasses depends greatly on the application of the system. In any case, an objective function needs to be defined. Such an objective function could for instance represent the amount of consumed energy, the amount of drag, the deviations of actuators from their preferred position, or even a combination of those. An optimisation method could then attempt to find the actuator set that minimises (or maximises) the objective function, based on actuator effectiveness matrix **H**. Several researches have already theoretically applied this concept. Both Härkegård [32] and Stolk [33] use constrained quadratic programming to solve the control allocation problem. Härkegård shows a hypothetical

example of an application on a Simulink based realistic and over-actuated fighter aircraft model. Stolk theoretically applies an INCA controller on the also over-actuated Innovative Control Effector or ICE aircraft. The previously mentioned Höppener [31] also uses quadratic programming in his INDI controller for the control of a conventional quadcopter, basically making it an INCA controller as well.

INCA deals with challenges named earlier. There is no need for the tuning of gains, its model dependency is supposedly low, and it is able to deal with the over-action of a quadplane. Its performance however depends heavily on the efficiency of the optimisation method, which needs to be quick enough for online use. A variety of these methods will be discussed in Section 4.

4 INCA Optimisation

A controller that uses INCA might be supposed to solve the allocation problem at a rate in the same order of magnitude as 100 Hz, resulting in a very short processing time per time step. This means the optimisation method used has to be rather efficient and reliable. Various surveys provide a clear overview of different optimisation methods that are relevant for control allocation, like Stolk [33], Bodson [34], Harkegard [35] and Johansen and Fossen [36]. This section is meant to elucidate methods from those surveys, found to be most promising for use in an INCA controller.

4.1. Generalised Inverse

When the control allocation problem is defined without actuator constraints, it can be solved by using a generalised inverse. A typical Control Allocation objective is to minimise the Weighted Least Squares or WLS control effort, defined as the difference between actual and desired actuator positions, while satisfying the control demand by some combination of actuator inputs:

$$\underset{\Delta\delta}{min}\,(\Delta\delta - \Delta\delta_p)^T \mathbf{W}(\Delta\delta - \Delta\delta_p) \tag{4.1a}$$

$$\text{subject to } \mathbf{H}\Delta\delta = \tau_c \tag{4.1b}$$

where δ_p is the vector containing the preferred actuator positions, and $\mathbf{W}$ is a weighting matrix to prioritise some actuators over others. Johansen and Fossen [36] state that as long as actuator effectiveness matrix $\mathbf{H}$ has full rank, the solution can be found by:

$$\Delta\delta = (\mathbf{I} - \mathbf{GH})\Delta\delta_p + \mathbf{G}\tau_c \tag{4.2a}$$

$$\text{where } \mathbf{G} = \mathbf{W}^{-1}\mathbf{H}^T(\mathbf{H}\mathbf{W}^{-1}\mathbf{H}^T)^{-1} \tag{4.2b}$$

When $\mathbf{W} = \mathbf{I}$ and $\delta_p = 0$, the optimisation problem and it solution simply to the following form:

$$\underset{\delta}{min}\,\frac{1}{2}\Delta\delta^T\Delta\delta \tag{4.3a}$$

$$\text{subject to } \mathbf{H}\Delta\delta = \tau_c \tag{4.3b}$$

$$\Delta\delta = \mathbf{H}^+\tau_c \tag{4.4a}$$

$$\text{where } \mathbf{H}^+ = \mathbf{H}^T(\mathbf{H}\mathbf{H}^T)^{-1} \tag{4.4b}$$

$\mathbf{H}^+$ is known as the Generalized Moore-Penrose or Psuedo-Inverse [37], and provides a simple algebraic solution to the optimisation problem defined in Equation 4.1a. The main disadvantage of this method is that

it doesn't allow any actuator constraints to be used, and therefore often yields unfeasible solutions. Other solvers are however often based on this method.

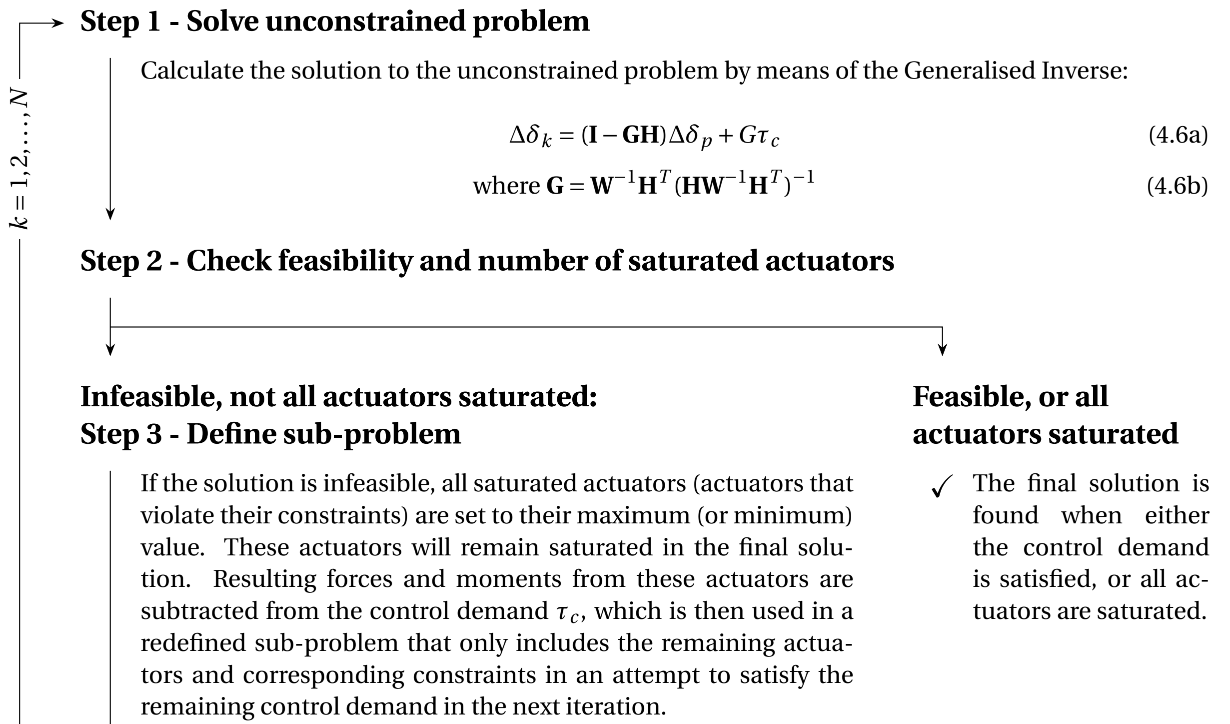
4.2. Redistributed Pseudo-Inverse

One solver that does handle actuator constraints uses a so-called Redistributed Pseudo-Inverse, or RPI for short, and is described by Virnig and Bodden [38]. The problem ould be defined as follows:

$$\underset{\Delta\delta}{min}\,(\Delta\delta - \Delta\delta_p)^T \mathbf{W}(\Delta\delta - \Delta\delta_p) \tag{4.5a}$$

$$\text{subject to } \mathbf{H}\Delta\delta = \tau_c \text{ and } \Delta\delta_{min} \leq \Delta\delta \leq \Delta\delta_{max} \tag{4.5b}$$

The RPI method consists of an iterative process that solves a number of unconstrained sub-problems by means of the previously described Generalised Inverse. It does so in the following manner:



The method is easy to implement, and it guarantees a feasible solution. According to Stolk [33] however, it is not guaranteed that this solver will find a solution that matches the control demand optimally.

4.3. Quadratic programming

When looking at control allocation solvers, three requirements need to be satisfied according to Buffington [39]: feasibility (the solution needs to be achievable), deficiency (if not achievable, the solution needs to be degraded to a achievable one) and sufficiency (there should only be one optimum). Section 4.1 shows that especially feasibility and deficiency requirements are not necessarily met by a generalised inverse. To change this, the WLS problem could be redefined to include the difference between control demand and the achieved moments, again having actuator limits as constraints:

$$\underset{\Delta\delta}{min}\,\|\mathbf{W}_\tau(\mathbf{H}\Delta\delta - \tau_c)\|_2 + \|\gamma\mathbf{W}_\delta(\Delta\delta_p - \Delta\delta)\|_2 \tag{4.7a}$$

$$\text{subject to } \delta_{min} \leq \delta \leq \delta_{max} \text{ and } \dot{\delta} \leq \dot{\delta}_{max} \tag{4.7b}$$

where $\mathbf{W}_\tau$ and $\mathbf{W}_\delta$ are weighting matrices, and γ is a scaling factor to prioritise one sub-objective over another, for instance control demand satisfaction versus minimum actuator deflection.. This type of objective function is called a *Quadratic Program*, and can include as many separate sub-objectives as one might need. Quadratic Programming is often used for Control Allocation problems. Härkegård [32] presents it as a suitable method, and Stolk [33] and Höppener [31] both apply it, on a modern fighter jet and a quadcopter UAV respecitvely. For easier processing, the objective function is often rewritten in the following manner:

$$\min_{\Delta\delta} \|\mathbf{W}_\tau(\mathbf{H}\Delta\delta - \tau_c)\|_2 + \|\gamma\mathbf{W}_\delta(\Delta\delta_p - \Delta\delta)\|_2 = \tag{4.8a}$$

$$\min_{\Delta\delta} \|\mathbf{F}\Delta\delta - g)\|_2 = \tag{4.8b}$$

$$\min_{\Delta\delta} \sqrt{\Delta\delta^T\mathbf{F}^T\mathbf{F}\Delta\delta + 2g\mathbf{F}\Delta\delta + g^Tg} = \tag{4.8c}$$

$$\min_{\Delta\delta} \Delta\delta^T\mathbf{Q}\Delta\delta + c^T\Delta\delta \tag{4.8d}$$

$$\text{where } \mathbf{F} = \begin{pmatrix}\mathbf{W}_\tau\mathbf{H}\\ \gamma\mathbf{W}_\delta\end{pmatrix},\ g = \begin{pmatrix}\mathbf{W}_\tau\tau_c\\ \gamma\mathbf{W}_\delta\Delta\delta_p\end{pmatrix},\ \mathbf{Q} = \mathbf{F}^T\mathbf{F} \text{ and } c = 2\mathbf{F}^Tg \tag{4.8e}$$

All actuator and actuator rate constraints can be captured in one expression, and adapted to their incremental form as follows:

$$\Delta\delta \le \delta_{max} - \delta_0,\ \Delta\delta \ge \delta_{min} - \delta_0,\ |\Delta\delta| \le \dot{\delta}_{max}\Delta t \rightarrow \mathbf{A}\Delta\delta \le b \tag{4.9a}$$

$$\text{where } \mathbf{A} = \begin{pmatrix}\mathbf{I}\\ -\mathbf{I}\end{pmatrix} \text{ and } b = \begin{pmatrix}\min(\delta_{max} - \delta_0, \dot{\delta}_{max}\Delta t)\\ -\max(\delta_{min} - \delta_0, -\dot{\delta}_{max}\Delta t)\end{pmatrix}, \tag{4.9b}$$

where δ_0 is the current actuator position. This results in a optimisation problem in a standardised quadratic form, with which many solvers can easily work:

$$\min_{\Delta\delta} \Delta\delta^T\mathbf{Q}\Delta\delta + c^T\Delta\delta \tag{4.10a}$$

$$\text{subject to } \mathbf{A}\Delta\delta \le b \tag{4.10b}$$

According to Gavin and Scruggs [40], when the inequality constraints are treated as equality constraints ($\mathbf{A} = b$ instead of $\mathbf{A} \le b$), the solution to the optimisation problem is given by the following linear system, as long as $\mathbf{Q}$ is a positive definite matrix and $\mathbf{A}$ has full row rank [36]:

$$\begin{bmatrix}\mathbf{Q} & \mathbf{A}^T\\ \mathbf{A} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\Delta\delta\\ \lambda\end{bmatrix} = \begin{bmatrix}-c\\ b\end{bmatrix} \tag{4.11}$$

where λ is also know as the vector containing the Lagrange multipliers. From this linear system, explicit solutions for both the optimal input increment $\Delta\delta$ and Lagrange multipliers λ can be derived algebraically to:

$$\Delta\delta = -\mathbf{Q}^{-1}(\mathbf{A}^T\lambda + c) \tag{4.12a}$$

$$\text{where } \lambda = -(\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^T)^{-1}(\mathbf{A}\mathbf{Q}^{-1}c + b) \tag{4.12b}$$

The signs of Lagrange multipliers will be used later to determine what constraints to release during the optimisation process, and whether the solution has already reached its optimum.

4.4. The Active Set Method

To deal with inequalities constraints like the ones needed for a typical control allocation problem, one often needs an iterative solver. One example of such a solver is the Active Set Method. Both Stolk [33] and Höppener [31] use it in their control allocation problems. A detailed description as provided by [35] is summarised below:

Step 1 - Choose feasible starting point

The solver's efficiency greatly depends on it's starting point. Often, the solution from the previous time step is used, since the solution typically does not change significantly between time steps in a control allocation problem.

Step 2 - Determine active set of constraints and redefine problem

All *active constraints*, i.e. constraints at which a control input saturates, will be treated as equality constraints, except those released from the active set by the previous iteration. All other constraints are being disregarded for the rest of the iteration. The redefined problem is now expressed as:

$$\min_{\Delta\delta} \Delta\delta^T \mathbf{Q} \Delta\delta + c^T \Delta\delta \tag{4.13a}$$

$$\text{subject to } \mathbf{A}_{act} = b_{act} \tag{4.13b}$$

where the subscript "act" indicates the active set of constraints.

Step 3 - Calculate Lagrange multipliers and solution to redefined problem

$k = 1, 2, \ldots, N$

The Lagrange multipliers and solution to the redefined problem are derived as follows:

$$\lambda_{act} = (\mathbf{A}_{act} \mathbf{Q}^{-1} \mathbf{A}_{act}{}^T)^{-1} (\mathbf{A}_{act} \mathbf{Q}^{-1} c + b_{act}) \tag{4.14}$$

$$\Delta\delta_k = \mathbf{Q}^{-1} (\mathbf{A}_{act}^T \lambda_{act} + c) \tag{4.15}$$

where the subscript "k" denotes the current iteration. The Lagrange multiplier vector λ contains one value for each of the active constraints.

Step 4 - Check feasibility

If solution is infeasible:

Correct the solution by taking the maximum relative step α from the previous to the new solution without losing feasibility:

$$\Delta\delta_k = \Delta\delta_{k-1} + \alpha(\Delta\delta_k - \Delta\delta_{k-1}) \tag{4.16}$$

$$\alpha = min((\Delta\delta_{viol_{k-1}} - b_{viol}) \oslash \Delta\delta_{viol_k}) \tag{4.17}$$

where the subscript "$viol$" denotes saturated control inputs and violated constraints, subscript "k" denotes the current iteration, subscript "$k-1$" denotes the previous iteration and $\oslash$ is an operator for Hadamard (element-wise) vector division [41].

Add the corresponding constraint to the active set of constraints.

If solution is feasible:

If all $\lambda \geq 0$:

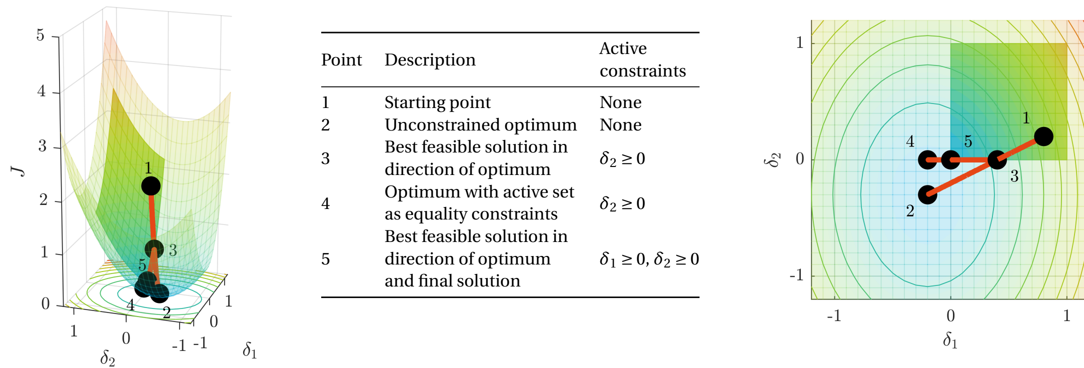
✓ The optimal solution is found.

If not all $\lambda \geq 0$:

Release constraint corresponding to the most negative value in λ from the active set.

Step 5 - Repeat iteration

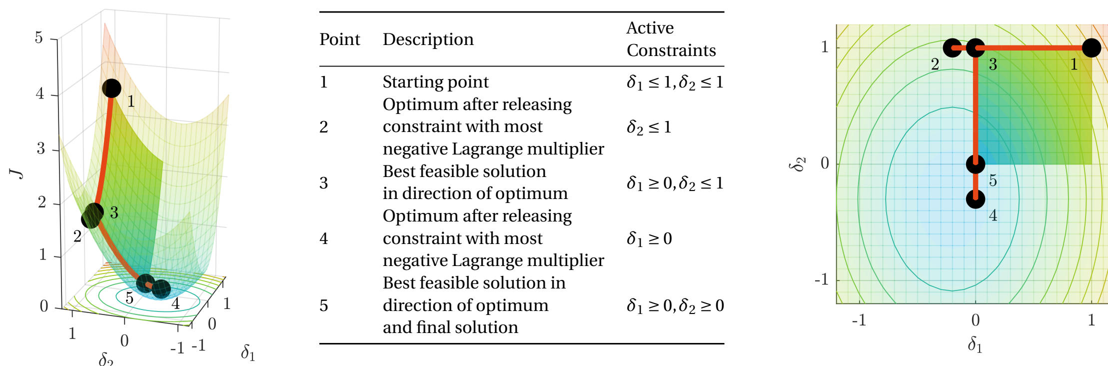
Repeat from Step 3 with the new active set of constraints and $\Delta\delta_k = \Delta\delta_{k-1}$



Point	Description	Active constraints
1	Starting point	None
2	Unconstrained optimum	None
3	Best feasible solution in direction of optimum	$\delta_2 \geq 0$
4	Optimum with active set as equality constraints	$\delta_2 \geq 0$
5	Best feasible solution in direction of optimum and final solution	$\delta_1 \geq 0, \delta_2 \geq 0$

Figure 4.1: The Active Set Method performed on a cost function J with a two-dimensional input space (Constraints: $0 \leq \delta_1 \leq 1$ and $0 \leq \delta_2 \leq 1$, starting point: $(\delta_1, \delta_2) = (0.8, 0.2)$)

An example of solving a two-dimensional input space cost function with the Active Set Method is given in Figure 4.1. It shows the procedure described above on a simplified example. It uses a starting point without any active constraints, and shows how maximum steps towards the optimum are taken in order to ensure feasibility. Another similar example is given in Figure 4.2. This example shows how constraints can become active or be released during iterations. Note that this example takes more computational effort, since twice releasing one of the constraints take two extra iterations with respect to the example in Figure 4.1. This shows, that choosing a suitable starting point has a significant effect on the solver's efficiency. In control allocation however, a smart starting point is always at hand, since each solution is likely to be in the neighbourhood of the solution of the previous time step.



Point	Description	Active Constraints
1	Starting point	$\delta_1 \leq 1, \delta_2 \leq 1$
2	Optimum after releasing constraint with most negative Lagrange multiplier	$\delta_2 \leq 1$
3	Best feasible solution in direction of optimum	$\delta_1 \geq 0, \delta_2 \leq 1$
4	Optimum after releasing constraint with most negative Lagrange multiplier	$\delta_1 \geq 0$
5	Best feasible solution in direction of optimum and final solution	$\delta_1 \geq 0, \delta_2 \geq 0$

Figure 4.2: The Active Set Method performed on a cost function J with a two-dimensional input space (Constraints: $0 \leq \delta_1 \leq 1$ and $0 \leq \delta_2 \leq 1$, starting point: $(\delta_1, \delta_2) = (1.0, 1.0)$)

The Active Set Method has a relatively low computational effort, especially when it starts at a well-chosen starting point. Furthermore, the solver's solution during each iteration always progresses towards the final solution of that time step. This means that chances are small that the solver will produce a very bad solution if cut off shortly. This results in the Active Set Method being very suitable for control allocation applications.

5 Further Research and Experiments

While Incremental Non-linear Control Allocation has proven itself in theory and simulations, it hasn't actually flown yet, let alone on a quadplane. The proposed Master thesis research objective is therefore:

> *...to asses the suitability of Incremental Non-linear Control Allocation (INCA) on the over-actuated TU Delft quadplane.*

The first part of the research focuses on assessing the suitability of an INCA controller to control all nine actuators of the TU Delft quadplane. Important factors are to research whether the controller is able to solve the over-actuation problem, and whether the cost functions can be selected in such a way that the most sensible solution is found. In order to also efficiently allocate control to the quadplane's tail rotor thrust and full quadcopter actuator set thrust, it might be interesting to include forces to the control demand vector used in the INCA controller. This is partially because the quadcopter actuator set is used for both thrust and attitude control. When the INCA optimisation only includes attitude control, and thrust is added later, saturation might occur. Taking both into consideration during the optimisation process should result in an optimal actuator output with minimal saturation.

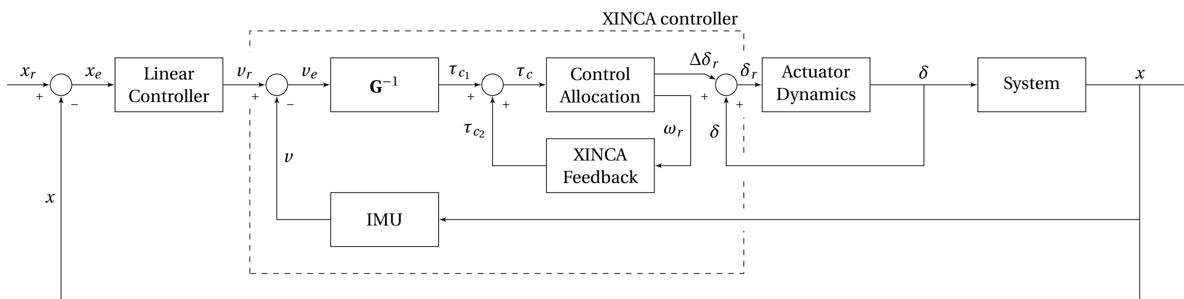


Figure 5.1: A schematic representation of an XINCA controller
(x = state vector, v = virtual input, τ = control demand, δ = control input vector)

A typical issue that should be resolved as well is that the quadplane sometimes has to choose between direct control of its accelerations by using an available actuator in that direction, or indirect control by first changing its attitude to deflect the thrust vector in the desired direction. An example would be to accelerate a quadplane forward directly like a fixed wing UAV by using its tail rotor, or indirectly like a quadcopter by pitching forward and increasing thrust. Making such a choice however typically happens in the linear controller that only tells the INCA controller to either pitch or move forward. In conventional quadplane controllers, this would simply depend on the actuator set being used at that particular moment. In order to simplify the linear controller, the INCA optimisation could possibly be adapted to include optimising attitude angles. This way, the linear controller only needs to tell the INCA controller to move forward, and the INCA controller can weigh the option of using the tail rotor against pitching forward and increasing thrust. Attitude optimisation cannot be accomplished in linear controller itself, because the optimal attitude highly depends on what

forces and moments the available actuators need to generate in certain directions. They should therefore be optimised in one combined optimisation process. A proposed name for a controller that does so is XINCA, or Extended Incremental Non-linear Control Allocation. A conceptual diagram for this innovative XINCA controller is shown in Figure 5.1. Here, the control allocation block calculates not only optimal control inputs, but also optimal attitude angels. This feedback is processed into an additional control demand, which is added to the original one.

These proposed focal points of the research translate into the following research questions:

Is INCA a suitable means for efficiently controlling the nine actuators of a quadplane?

- *Does it solve the over-actuation problem?*
- *Does it provide sensible solutions in case of actuator saturation?*

Is INCA able to work with a control demand that includes both angular and linear accelerations?

Can INCA optimise both attitude and position loops at the same time, by including the quadplane's attitude angles in the control input vector?

These questions should be addressed by a theoretical analysis of the proposed methods, combined with both simulations of quadplane flights with specific control demands and physical test flights, performed on the TU Delft quadplane. The proposed research planning is pictured in Figure 5.2.

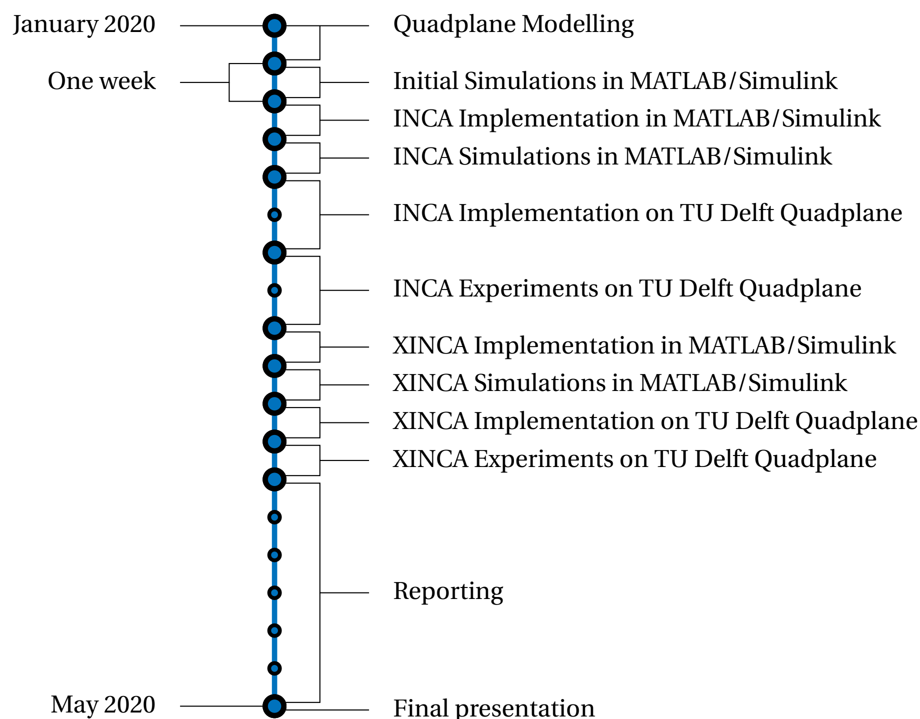


Figure 5.2: Research planning

6 Conclusion

The TU Delft quadplane is an over-actuated aircraft with both quadcopter and fixed-wing actuator sets. Similar aircraft are often controlled in two distinct flight modes, in which they either behave like a conventional quadcopter or a conventional fixed-wing aircraft. This way, occasional actuator saturation will not be compensated by available redundant actuators. Also, without proper control allocation that considers all available actuators at all times, the quadplane might not always choose the most efficient actuators available. When flying forward in quadcopter mode, it will pitch forward and increase thrust instead of more efficiently using its tail rotor.

Incremental Non-linear Control Allocation, or INCA for short, seems to be a suitable way of controlling the TU Delft quadplane. It is supposed to deal with both over-actuation and saturation, and allocate control to different actuators in an efficient manner. The allocation process is driven by a optimisation process, that is repeated for every time step. Literature shows the Active Set Method to be quite capable of solving control allocation problems at a high rate. It can optimise a multi-objective Quadratic Program, and is therefore able to not only satisfy a control demand as well as possible, but also to minimise control effort. The Active Set Method has the property of always improving its solution during its iterative process. This ensures that the best solution found so far is always at hand, might the process be terminated prematurely. Literature shows that the Active Set Method especially performs well if its initial guess is chosen well. In control allocation, the different solutions between time steps can be assumed to vary only slightly over time. This way, the solution of a previous time step is likely to be a very good estimation of the solution of the next time step.

The proposed research is focused on further assessing the suitability of INCA for use on the TU Delft quadplane. Important focal points are to determine whether such a controller actually solves the quadplane's overaction problem, provides sensible solutions in case of actuator saturation, and whether linear body forces can be included in the linear controller's control demand. Also, a extension of the current INCA controller to include attitude angles in its optimisation is proposed to be investigated. This controller would be named XINCA, or Extended Incremental Non-linear Control Allocation. Both numerical simulations and test flights should be performed in order to prove the hypotheses.

Bibliography

[1] Bai Zhiqiang, Liu Peizhi, Wang Jinhua, and Hu Xiongwen. Simulation system design of a uav helicopter. *2011 International Conference on Electric Information and Control Engineering*, 2011. doi: 10.1109/iceice.2011.5778108.

[2] Teguh Santoso Lembono, Jun En Low, Luke Soe Thura Win, Shaohui Foong, and U-Xuan Tan. Orientation filter and angular rates estimation in monocopter using accelerometers and magnetometer with the extended kalman filter. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017. doi: 10.1109/icra.2017.7989527.

[3] Teppo Luukkonen. Modelling and control of quadcopter, Aug 2011.

[4] E.J.J. Smeur, D.C. Höppener, and C. De Wagter. Prioritized control allocation for quadrotors subject to saturation. *International Micro Air Vehicle Conference and Flight Competition (IMAV)*, 2017.

[5] Francesco Forte, Roberto Naldi, Andrea Serrani, and Lorenzo Marconi. Control of modular aerial robots: Combining under- and fully-actuated behaviors. *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, 2012. doi: 10.1109/cdc.2012.6425886.

[6] Yan Ma, Zhihao Cai, Ningjun Liu, and Yingxun Wang. System composition and longitudinal motion control simulation of vehicular towed autogyro. *2016 IEEE Chinese Guidance, Navigation and Control Conference (CGNCC)*, 2016. doi: 10.1109/cgncc.2016.7828926.

[7] Yoshiyuki Higashi, Takanori Emaru, Kazuo Tanaka, and Hua Wang. Development of a cyclogyro-based flying robot with variable attack angle mechanisms. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006. doi: 10.1109/iros.2006.282435.

[8] Jacob Apkarian. Attitude control of pitch-decoupled vtol fixed wing tiltrotor. *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2018. doi: 10.1109/icuas.2018.8453473.

[9] Ryuta Takeuchi, Keigo Watanabe, and Isaku Nagai. Development and control of tilt-wings for a tilt-type quadrotor. *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*, 2017. doi: 10.1109/icma.2017.8015868.

[10] Christophe De Wagter and Ewoud J.J Smeur. Control of a hybrid helicopter with wings. *International Journal of Micro Air Vehicles*, 9(3):209–217, Nov 2017. doi: 10.1177/1756829317702674.

[11] Matthew E. Argyle, Jason M. Beach, Randal W. Beard, Timothy W. Mclain, and Stephen Morris. Quaternion based attitude error for a tailsitter in hover flight. *2014 American Control Conference*, 2014. doi: 10.1109/acc.2014.6859324.

[12] Danial Sufiyan Bin Shaiful, Luke Thura Soe Win, Jun En Low, Shane Kyi Hla Win, Gim Song Soh, and Shaohui Foong. Optimized transition path of a transformable hovering rotorcraft (thor). *2018 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2018. doi: 10.1109/aim.2018.8452703.

[13] Janith Kalpa Gunarathna and Rohan Munasinghe. Development of a quad-rotor fixed-wing hybrid unmanned aerial vehicle. *2018 Moratuwa Engineering Research Conference (MERCon)*, 2018. doi: 10.1109/mercon.2018.8421941.

[14] Jian Zhang, Zhiming Guo, and Liaoni Wu. Research on control scheme of vertical take-off and landing fixed-wing uav. *2017 2nd Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*, 2017. doi: 10.1109/acirs.2017.7986093.

[15] David Orbea, Jessica Moposita, Wilbert G. Aguilar, Manolo Paredes, Rolando P. Reyes, and Luis Montoya. Vertical take off and landing with fixed rotor. *2017 CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, 2017. doi: 10.1109/chilecon.2017.8229691.

[16] Ma Tielin, Yang Chuanguang, Gan Wenbiao, Xue Zihan, Zhang Qinling, and Zhang Xiaoou. Analysis of technical characteristics of fixed-wing vtol uav. *2017 IEEE International Conference on Unmanned Systems (ICUS)*, 2017. doi: 10.1109/icus.2017.8278357.

[17] Gerardo Flores and R. Lozano. Lyapunov-based controller using singular perturbation theory: An application on a mini-uav. *2013 American Control Conference*, 2013. doi: 10.1109/acc.2013.6580063.

[18] Zhang Daibing, Wang Xun, and Kong Weiwei. Autonomous control of running takeoff and landing for a fixed-wing unmanned aerial vehicle. *2012 12th International Conference on Control Automation Robotics & Vision (ICARCV)*, 2012. doi: 10.1109/icarcv.2012.6485292.

[19] Marco Palermo and Roelof Vos. Experimental aerodynamic analysis of a 4.6%-scale flying-v subsonic transport. *AIAA Scitech 2020 Forum*, May 2020. doi: 10.2514/6.2020-2228.

[20] D. Jin Lee, Byoung-Mun Min, Min-Jea Tahk, Hyochoong Bang, and D.h Shim. Autonomous flight control system design for a blended wing body. *2008 International Conference on Control, Automation and Systems*, 2008. doi: 10.1109/iccas.2008.4694548.

[21] Z. Liu, S. Tang, M. Li, and J. Guo. Optimal control of thrust-vectored vtol uav in high-manoeuvering transition flight. *The Aeronautical Journal*, 122(1250):598–619, 2018. doi: 10.1017/aer.2018.1.

[22] G. C. H. E. De Croon, M. Perçin, B.D.W. Remes, R. Ruijsink, and C. De Wagter. *The DelFly: design, aerodynamics, and artificial intelligence of a flapping wing robot.* Springer, 2016.

[23] Debora Grant and James Rand. The balloon assisted launch system - a heavy lift balloon. *International Balloon Technology Conference*, 1999. doi: 10.2514/6.1999-3872.

[24] S. Bermudez I Badia, P. Pyk, and P.f.m.j. Verschure. A biologically based flight control system for a blimp-based uav. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005. doi: 10.1109/robot.2005.1570579.

[25] Lilium, 2020. URL `https://lilium.com/`.

[26] Ehang|official site-drones anyone can fly, 2020. URL `http://www.ehang.com/`.

[27] Anthony Aman. Professional insertion internship report, Sep 2017.

[28] Tom Van Dijk. Crash course paparazzi 2019 autonomous flight with paparazzi - part 3, Feb 2019. URL `https://github.com/tudelft/coursePaparazzi/raw/master/part3_autonomous_flight_with_paparazzi.pdf`.

[29] Mituhiko Araki. Pid control. *Control Systems, Robotics, and Automation*, 2:58–79, Oct 2009.

[30] Joseph Horn. Non-linear dynamic inversion control design for rotorcraft. *Aerospace*, 6(3):38, 2019. doi: 10.3390/aerospace6030038.

[31] D.C. Höppener. Actuator saturation handling using weighted optimal control allocation applied to an indi controlled quadcopter. Master's thesis, Delft University of Technology, 2017.

[32] Ola Härkegård. Dynamic control allocation using constrained quadratic programming. *AIAA Guidance, Navigation, and Control Conference and Exhibit*, May 2002. doi: 10.2514/6.2002-4761.

[33] A.R.J. Stolk. Minimum drag control allocation for the innovative control effector aircraft. Master's thesis, Delft University of Technology, 2017.

[34] Marc Bodson. Evaluation of optimization methods for control allocation. *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Jun 2001. doi: 10.2514/6.2001-4223.

[35] O. Harkegard. Efficient active set algorithms for solving constrained least squares problems in aircraft control allocation. *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, 2002. doi: 10.1109/cdc.2002.1184694.

[36] Tor A. Johansen and Thor I. Fossen. Control allocation—a survey. *Automatica*, 49(5):1087–1103, 2013. doi: 10.1016/j.automatica.2013.01.035.

[37] K. Manjunatha Prasad and R.b. Bapat. The generalized moore-penrose inverse. *Linear Algebra and its Applications*, 165:59–69, 1992. doi: 10.1016/0024-3795(92)90229-4.

[38] John Virnig and David Bodden. Multivariable control allocation and control law conditioning when control effectors limit. *Guidance, Navigation, and Control Conference*, 1994. doi: 10.2514/6.1994-3609.

[39] James F. Buffington. Modular control law design for the innovative control effectors (ice) tailless fighter aircraft configuration 101-3. *AIAA Guidance, Navigation and Control Conference*, Jan 1999. doi: 10.21236/ada374954.

[40] Henri P. Gavin and Jeffrey T. Scruggs. Constrained optimization using lagrange multipliers. *CEE 201L. Uncertainty, Design, and Optimization*, 2020.

[41] Gordon Wetzstein, Douglas Lanman, Matthew Hirsch, and Ramesh Raskar. Tensor displays. *ACM Transactions on Graphics*, 31(4):1–11, Jan 2012. doi: 10.1145/2185520.2185576.

III

Part III: Appendices

A

Overview of simulations and flight experiments

	Description	Inner loop controller	Outer loop controller	Number of used actuators	Maximum actuator PWM pulse width [ms]	Flight designation
Simulations	Vertical takeoff and landing	INCA	XINCA	5	1900	SIM_TOL
	Vertical takeoff and landing with actuator saturation	INCA	XINCA	5	1310	SIM_SAT
	Forward and backwards flight	INCA	XINCA	5	1900	SIM_FWD
	First forward only flight for comparison	INCA	INDI	5	1900	SIM_COMP_INDI
	Second forward only flight for comparison	INCA	XINCA	5	1900	SIM_COMP_XINCA
Flight experiments	Vertical takeoff and landing	INCA	PID	7	1900	FLIGHT_TOL
	Vertical takeoff and landing with actuator saturation	INCA	PID	5	1460	FLIGHT_SAT
	Forward and backwards flight	INCA	XINCA	5	1900	FLIGHT_FWD

Table A.1: Overview of simulations and flight experiments

B

Paparazzi Airframe Configuration File

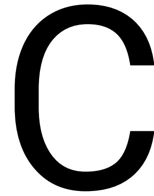
```
1  <!DOCTYPE airframe SYSTEM "../airframe.dtd">
2
3  <airframe name="Quadplane lisa_mx_2.1 pwm with XINCA and INDI on actuators excluding control surfaces">
4
5      <description>
6          Quadplane:
7          - modified mini-talon with cobra 2814/16 Kv1050 running a 10x8 prop
8          - quad with KISS esc + cobra 2207/2450kv running 6x3 props
9          - This quadplane runs XINCA on its outer loop and INDI in its inner loop without using
10           its control surface in order to minimise computational effort
11     </description>
12
13     <!-- FIRMWARE -->
14
15     <firmware name="rotorcraft">
16
17         <target name="ap" board="lisa_mx_2.1">
18             <!-- MPU6000 is configured to output data at 2kHz, but polled at 512Hz PERIODIC_FREQUENCY -->
19             <module name="radio_control" type="spektrum">
20                 <define name="RADIO_MODE"                        value="RADIO_GEAR"/>
21                 <define name="RADIO_KILL_SWITCH"                 value="RADIO_AUX1"/>
22                 <define name="USE_KILL_SWITCH_FOR_MOTOR_ARMING" value="1" />
23             </module>
24         </target>
25
26         <target name="nps" board="pc">
27             <module name="fdm" type="jsbsim"/>
28             <module name="radio_control" type="datalink"/>
29         </target>
30
31         <module name="actuators" type="pwm">
32             <define name="SERVO_HZ" value="400"/>
33             <define name="USE_SERVOS_7AND8"/>
34         </module>
35
36         <module name="telemetry" type="xbee_api"/>
37         <module name="imu" type="aspirin_v2.2"/>
38         <module name="gps" type="datalink"/>
39
40         <module name="stabilization" type="inca">
41             <define name="INDI_RPM_FEEDBACK"    value="FALSE"/>
42         </module>
43
44         <module name="guidance" type="xinca"/>
45
46         <module name="ahrs" type="int_cmpl_quat">
47             <configure  name="USE_MAGNETOMETER"                value="FALSE"/>
48             <define     name="AHRS_GRAVITY_HEURISTIC_FACTOR"   value="0"/>
49         </module>
50
```

```
51          <module name="ins" type="extended"/>
52          <module name="geo_mag"/>
53          <module name="sys_mon"/>
54
55      </firmware>
56
57      <!-- COMMANDS -->
58
59      <commands>
60          <axis name="ROLL"          failsafe_value="0"/>
61          <axis name="PITCH"         failsafe_value="0"/>
62          <axis name="YAW"           failsafe_value="0"/>
63          <axis name="THRUST"        failsafe_value="0"/>
64      </commands>
65
66      <!-- SERVOS -->
67
68      <servos driver="Pwm">
69          <servo name="FRONT_LEFT"         no="0" min="1000" neutral="1100" max="1900"/>
70          <servo name="FRONT_RIGHT"        no="1" min="1000" neutral="1100" max="1900"/>
71          <servo name="BACK_RIGHT"         no="2" min="1000" neutral="1100" max="1900"/>
72          <servo name="BACK_LEFT"          no="3" min="1000" neutral="1100" max="1900"/>
73          <servo name="AILERONS"           no="4" min="1000" neutral="1450" max="1900"/>
74          <servo name="RUDDERVATOR_LEFT"   no="5" min="1000" neutral="1450" max="1900"/>
75          <servo name="RUDDERVATOR_RIGHT"  no="6" min="1000" neutral="1450" max="1900"/>
76          <servo name="FLYMOTOR"           no="7" min="1000" neutral="1100" max="1900"/>
77      </servos>
78
79      <!-- COMMANDS LAWS -->
80
81      <command_laws>
82
83          <!-- Quadcopter actuatortor set -->
84          <set servo="FRONT_LEFT"     value="autopilot_get_motors_on() ? actuators_pprz[0] : -MAX_PPRZ"/>
85          <set servo="FRONT_RIGHT"    value="autopilot_get_motors_on() ? actuators_pprz[1] : -MAX_PPRZ"/>
86          <set servo="BACK_RIGHT"     value="autopilot_get_motors_on() ? actuators_pprz[2] : -MAX_PPRZ"/>
87          <set servo="BACK_LEFT"      value="autopilot_get_motors_on() ? actuators_pprz[3] : -MAX_PPRZ"/>
88
89          <!-- Fixed wing actuator set -->
90          <set servo="AILERONS"           value="autopilot_get_motors_on() ? actuators_pprz[4] : 0"/>
91          <set servo="RUDDERVATOR_LEFT"   value="autopilot_get_motors_on() ? actuators_pprz[5] : 0"/>
92          <set servo="RUDDERVATOR_RIGHT"  value="autopilot_get_motors_on() ? actuators_pprz[6] : 0"/>
93
94          <!-- Tail rotor -->
95          <set servo="FLYMOTOR"           value="(autopilot_get_motors_on() && -stateGetPositionNed_f()->z
        >= GUIDANCE_XINCA_H_THRES) ? actuators_pprz[4] : -MAX_PPRZ"/>
96
97      </command_laws>
98
99      <!-- SETTINGS -->
100
101     <section name="IMU" prefix="IMU_">
102
103         <!-- Replace this with your own calibration -->
104         <define name="MAG_X_NEUTRAL"        value="-102"/>
105         <define name="MAG_Y_NEUTRAL"        value="-69"/>
106         <define name="MAG_Z_NEUTRAL"        value="96"/>
107         <define name="MAG_X_SENS"           value="3.7163763222485424"  integer="16"/>
108         <define name="MAG_Y_SENS"           value="3.667894315633858"   integer="16"/>
109         <define name="MAG_Z_SENS"           value="4.040080277772171"   integer="16"/>
110
111         <define name="BODY_TO_IMU_PHI"      value="0."                  unit="deg"/>
112         <define name="BODY_TO_IMU_THETA"    value="0."                  unit="deg"/>
113         <define name="BODY_TO_IMU_PSI"      value="180."                unit="deg"/>
114
115     </section>
116
117     <section name="AHRS" prefix="AHRS_">
118
119         <!-- Delft magnetic field -->
120         <define name="H_X"                          value="0.39049610"/>
```

```
121         <define name="H_Y"                              value="0.00278894"/>
122         <define name="H_Z"                              value="0.92060036"/>
123         <define name="USE_GPS_HEADING"                  value="1"/>
124         <define name="HEADING_UPDATE_GPS_MIN_SPEED"     value="0"/>
125
126         <!-- For vibrating airfames -->
127         <define name="GRAVITY_HEURISTIC_FACTOR"         value="0"/>
128
129     </section>
130
131     <section name="INS" prefix="INS_">
132         <define name="USE_GPS_ALT"              value="1"/>
133         <define name="USE_GPS_ALT_SPEED"        value="1"/>
134     </section>
135
136     <section name="STABILIZATION_ATTITUDE" prefix="STABILIZATION_ATTITUDE_">
137
138         <!-- Setpoints -->
139         <define name="SP_MAX_PHI"     value="45."      unit="deg"/>
140         <define name="SP_MAX_THETA"   value="45."      unit="deg"/>
141         <define name="SP_MAX_R"       value="90."      unit="deg/s"/>
142         <define name="DEADBAND_A"     value="0"/>
143         <define name="DEADBAND_E"     value="0"/>
144         <define name="DEADBAND_R"     value="10"/>
145
146         <!-- Reference -->
147         <define name="REF_OMEGA_P"    value="400"      unit="deg/s"/>
148         <define name="REF_ZETA_P"     value="0.85"/>
149         <define name="REF_MAX_P"      value="400."     unit="deg/s"/>
150         <define name="REF_MAX_PDOT"   value="RadOfDeg(8000.)"/>
151
152         <define name="REF_OMEGA_Q"    value="400"      unit="deg/s"/>
153         <define name="REF_ZETA_Q"     value="0.85"/>
154         <define name="REF_MAX_Q"      value="400."     unit="deg/s"/>
155         <define name="REF_MAX_QDOT"   value="RadOfDeg(8000.)"/>
156
157         <define name="REF_OMEGA_R"    value="250"      unit="deg/s"/>
158         <define name="REF_ZETA_R"     value="0.85"/>
159         <define name="REF_MAX_R"      value="180."     unit="deg/s"/>
160         <define name="REF_MAX_RDOT"   value="RadOfDeg(1800.)"/>
161
162     </section>
163
164     <section name="STABILIZATION_ATTITUDE_INCA" prefix="STABILIZATION_INDI_">
165
166         <!-- Virtual input and actuator vector dimensions -->
167         <define name="INDI_OUTPUTS"       value="4"/>
168         <define name="INDI_NUM_ACT"       value="7"/>
169
170         <!-- Actuator properties -->
171         <define name="ACT_DYN_TAU"        value="{ 29.0,   29.0,   29.0,   29.0,  100.0,  100.0,  100.0 }"/>
172         <define name="ACT_IS_SERVO"       value="{    0,      0,      0,      0,      1,      1,      1 }"/>
173         <define name="ACT_IS_THRUST"      value="{    1,      1,      1,      1,      0,      0,      0 }"/>
174         <define name="ACT_IS_SURFACE"     value="{    0,      0,      0,      0,      1,      1,      1 }"/>
175         <define name="ACT_RATE_LIMIT"     value="{ 9600,   9600,   9600,   9600,   9600,   9600,   9600 }"/>
176         <define name="ACT_PREF"           value="{    0,      0,      0,      0,      0,      0,      0 }"/>
177         <define name="ACT_PRIORITIES"     value="{   10,     10,     10,     10,      1,      1,      1 }"/>
178
179         <!-- Control effectiveness -->
180         <define name="G1_ROLL"            value="{ 11.0,  -11.0,  -11.0,   11.0,   0.15,    0.0,    0.0 }"/>
181         <define name="G1_PITCH"           value="{  9.0,    9.0,   -9.0,   -9.0,    0.0,   0.11,  -0.11}"/>
182         <define name="G1_YAW"             value="{ -0.60,  0.60,  -0.60,   0.60,    0.0,  -0.03,  -0.03}"/>
183         <define name="G1_THRUST"          value="{ -0.80, -0.80,  -0.80,  -0.80,    0.0,    0.0,    0.0 }"/>
184
185         <!-- Counter torque effect of spinning up a rotor -->
186         <define name="G2"                 value="{-55.0,   55.0,  -55.0,   55.0,    0.0,    0.0,    0.0 }"/>
187
188         <!-- Reference acceleration for attitude control -->
189         <define name="REF_ERR_P"          value="200.0"/>
190         <define name="REF_ERR_Q"          value="200.0"/>
191         <define name="REF_ERR_R"          value="200.0"/>
```

```
192         <define name="REF_RATE_P"        value="28.0"/>
193         <define name="REF_RATE_Q"        value="28.0"/>
194         <define name="REF_RATE_R"        value="50.0"/>
195 
196         <define name="ESTIMATION_FILT_CUTOFF"    value="4.0"/>
197         <define name="FILT_CUTOFF"               value="5.0"/>
198 
199         <!-- Adaptive Learning Rate -->
200         <define name="USE_ADAPTIVE"      value="FALSE"/>
201         <define name="ADAPTIVE_MU"       value="0.00001"/>
202 
203         <!--Priority for each axis (roll, pitch, yaw and thrust)-->
204         <define name="WLS_PRIORITIES"    value="{100, 100, 10, 1000}"/>
205 
206         <!--Run every nth cycle -->
207         <define name="NTH_CYCLE"         value="1"/>
208 
209     </section>
210 
211     <section name="GUIDANCE_V" prefix="GUIDANCE_V_">
212         <define name="HOVER_KP"                  value="150"/>
213         <define name="HOVER_KD"                  value="80"/>
214         <define name="HOVER_KI"                  value="20"/>
215         <define name="NOMINAL_HOVER_THROTTLE"    value="0.5"/>
216         <define name="ADAPT_THROTTLE_ENABLED"    value="TRUE"/>
217     </section>
218 
219     <section name="GUIDANCE_H" prefix="GUIDANCE_H_">
220         <define name="MAX_BANK"          value="20" unit="deg"/>
221         <define name="USE_SPEED_REF"     value="TRUE"/>
222         <define name="PGAIN"             value="50"/>
223         <define name="DGAIN"             value="100"/>
224         <define name="AGAIN"             value="70"/>
225         <define name="IGAIN"             value="20"/>
226     </section>
227 
228     <section name="GUIDANCE_XINCA" prefix="GUIDANCE_XINCA_">
229 
230         <define name="OUTPUTS"           value="3"/>
231         <define name="NUM_ACT"           value="4"/>
232 
233         <define name="ACT_X_TAU"         value="60"/>
234         <define name="ACT_Z_TAU"         value="29"/>
235         <define name="U_PREF"            value="{0, 0, 0, 0}"/>
236         <define name="W_ACC"             value="{1, 1, 1}"/>
237         <define name="W_ACT"             value="{10, 10, 10, 1}"/>
238         <define name="GAMMA"             value="10000"/>
239         <define name="H_THRES"           value="0.3"/>
240 
241         <define name="C_L_ALPHA"         value="1.55"/>
242         <define name="C_X_TAIL_ROTOR"    value="2"/>
243         <define name="C_Z_THRUST"        value="-3.2"/>
244         <define name="MASS"              value="3.0"/>
245         <define name="WING_SURFACE"      value="0.24"/>
246 
247         <!--Run every nth cycle -->
248         <define name="NTH_CYCLE"         value="1"/>
249 
250     </section>
251 
252     <section name="NAV">
253         <define name="ARRIVED_AT_WAYPOINT" value="0.2" unit="m"/>
254     </section>
255 
256     <section name="BAT">
257         <define name="CATASTROPHIC_BAT_LEVEL"        value="13.2" unit="V"/>
258         <define name="CRITIC_BAT_LEVEL"              value="14.0" unit="V"/>
259         <define name="LOW_BAT_LEVEL"                 value="14.8" unit="V"/>
260         <define name="MAX_BAT_LEVEL"                 value="16.8" unit="V"/>
261         <define name="MILLIAMP_AT_FULL_THROTTLE"     value="30000"/>
262     </section>
```

```
263
264     <section name="SIMULATOR" prefix="NPS_">
265
266         <define name="ACTUATOR_NAMES"    value="front_motor, right_motor, back_motor, left_motor, ailerons,
          left_ruddervator, right_ruddervator, tail_rotor" type="string[]"/>
267         <define name="JSBSIM_MODEL"      value="quadplane"                          type="string"/>
268         <define name="SENSORS_PARAMS"    value="nps_sensors_params_default.h"     type="string"/>
269         <define name="NO_MOTOR_MIXING"   value="TRUE"/>
270
271         <!-- Mode switch on joystick ch5 (numbering starts at zero) -->
272         <define name="JS_AXIS_MODE"      value="4"/>
273
274     </section>
275
276     <section name="AUTOPILOT">
277         <define name="MODE_MANUAL"           value="AP_MODE_RC_DIRECT"/>
278         <define name="MODE_AUTO1"            value="AP_MODE_ATTITUDE_DIRECT"/>
279         <define name="MODE_AUTO2"            value="AP_MODE_NAV"/>
280         <define name="NO_RC_THRUST_LIMIT"    value="TRUE"/>
281     </section>
282
283 </airframe>
```



Source code of the INCA module

This code is based on an existing Paparazzi UAV inner loop INDI module by Smeur et al. [1].

[1] Ewoud J.J. Smeur, Qiping Chu, and Guido C.H.E. De Croon. Adaptive incremental nonlinear dynamic inversion for attitude control of micro air vehicles. 2015. doi: https://doi.org/10.2514/1.G001490.


```
/*
 * Original module: Copyright (C) 2015 Ewoud Smeur <ewoud.smeur@gmail.com>
 * Quadplane extension: Copyright (C) 2020 Jan Karssies <hjkarssies@gmail.com>
 * MAVLab Delft University of Technology
 *
 * This file is part of paparazzi.
 *
 * paparazzi is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * paparazzi is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with paparazzi; see the file COPYING.  If not, write to
 * the Free Software Foundation, 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/** @file stabilization_attitude_quat_indi.c
 * @brief MAVLab Delft University of Technology
 * This control algorithm is Incremental Nonlinear Dynamic Inversion (INDI)
 *
 * This original model is an implementation of the publication in the
 * journal of Control Guidance and Dynamics: Adaptive Incremental Nonlinear
 * Dynamic Inversion for Attitude Control of Micro Aerial Vehicles
 * http://arc.aiaa.org/doi/pdf/10.2514/1.G001490
 *
 * The adapation is based on the following master thesis:
 * Extended Nonlinear Control Allocation on the TU Delft Quadplane - H.J. Karssies
 * http://repository.tudelft.nl/
 */

#include "firmwares/rotorcraft/stabilization/stabilization_inca.h"
#include "firmwares/rotorcraft/stabilization/stabilization_attitude.h"
#include "firmwares/rotorcraft/stabilization/stabilization_attitude_rc_setpoint.h"
#include "firmwares/rotorcraft/stabilization/stabilization_attitude_quat_transformations.h"

#include "math/pprz_algebra_float.h"
#include "state.h"
#include "generated/airframe.h"
```

```
46 #include "subsystems/radio_control.h"
47 #include "subsystems/actuators.h"
48 #include "subsystems/abi.h"
49 #include "filters/low_pass_filter.h"
50 #include "wls/wls_alloc.h"
51 #include <stdio.h>
52
53 // Factor that the estimated G matrix is allowed to deviate from initial one
54 #define INDI_ALLOWED_G_FACTOR 2.0
55
56 float du_min[INDI_NUM_ACT];
57 float du_max[INDI_NUM_ACT];
58 float du_pref[INDI_NUM_ACT];
59 float indi_v[INDI_OUTPUTS];
60 float indi_v_abs[INDI_OUTPUTS];
61 float *Bwls[INDI_OUTPUTS];
62 int num_iter = 0;
63 int num_cycle = 1; // Cycle count (resets every nth cycle)
64 #ifdef STABILIZATION_INDI_NTH_CYCLE
65 int run_nth_cycle = STABILIZATION_INDI_NTH_CYCLE; // Run every nth cycle
66 #else
67 int run_nth_cycle = 1; // Run every nth cycle
68 #endif
69 struct FloatVect3 speed_body;
70
71 static void lms_estimation(void);
72 static void get_actuator_state(void);
73 static void calc_g1_element(float dx_error, int8_t i, int8_t j, float mu_extra);
74 static void calc_g2_element(float dx_error, int8_t j, float mu_extra);
75 static void calc_g1g2_pseudo_inv(void);
76 static void bound_g_mat(void);
77 static void scale_surface_effectiveness(void);
78
79 int32_t stabilization_att_indi_cmd[COMMANDS_NB];
80 struct ReferenceSystem reference_acceleration = {
81   STABILIZATION_INDI_REF_ERR_P,
82   STABILIZATION_INDI_REF_ERR_Q,
83   STABILIZATION_INDI_REF_ERR_R,
84   STABILIZATION_INDI_REF_RATE_P,
85   STABILIZATION_INDI_REF_RATE_Q,
86   STABILIZATION_INDI_REF_RATE_R,
87 };
88
89 #if STABILIZATION_INDI_USE_ADAPTIVE
90 bool indi_use_adaptive = true;
91 #else
92 bool indi_use_adaptive = false;
93 #endif
94
95 #ifdef STABILIZATION_INDI_ACT_RATE_LIMIT
96 float act_rate_limit[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_RATE_LIMIT;
97 #endif
98
99 #ifdef STABILIZATION_INDI_ACT_IS_SERVO
100 bool act_is_servo[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_IS_SERVO;
101 #else
102 bool act_is_servo[INDI_NUM_ACT] = {0};
103 #endif
104
105 #ifdef STABILIZATION_INDI_ACT_IS_THRUST
106 bool act_is_thrust[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_IS_THRUST;
107 #else
108 bool act_is_thrust[INDI_NUM_ACT] = {1};
109 #endif
110
111 #ifdef STABILIZATION_INDI_ACT_IS_SURFACE
112 bool act_is_surface[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_IS_SURFACE;
113 #else
114 bool act_is_surface[INDI_NUM_ACT] = {0};
115 #endif
116
```

```
#ifdef STABILIZATION_INDI_ACT_PREF
// Preferred (neutral, least energy) actuator value
float act_pref[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_PREF;
#else
// Assume 0 is neutral
float act_pref[INDI_NUM_ACT] = {0.0};
#endif

float act_dyn_tau[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_DYN_TAU;
float act_dyn[INDI_NUM_ACT];

/** Maximum rate you can request in RC rate mode (rad/s)*/
#ifndef STABILIZATION_INDI_MAX_RATE
#define STABILIZATION_INDI_MAX_RATE 6.0
#endif

#ifdef STABILIZATION_INDI_WLS_PRIORITIES
static float Wv[INDI_OUTPUTS] = STABILIZATION_INDI_WLS_PRIORITIES;
#else
//State prioritization {W Roll, W pitch, W yaw, TOTAL THRUST}
static float Wv[INDI_OUTPUTS] = {1000, 1000, 1, 100};
#endif

#ifdef STABILIZATION_INDI_ACT_PRIORITIES
static float Wu[INDI_NUM_ACT] = STABILIZATION_INDI_ACT_PRIORITIES;
#else
//State prioritization {W Roll, W pitch, W yaw, TOTAL THRUST}
static float Wu[INDI_NUM_ACT] = {[0 ... INDI_NUM_ACT] = 1};
#endif

// variables needed for control
float actuator_state_filt_vect[INDI_NUM_ACT];
struct FloatRates angular_accel_ref = {0., 0., 0.};
float angular_acceleration[3] = {0., 0., 0.};
float actuator_state[INDI_NUM_ACT];
float indi_u[INDI_NUM_ACT];
float indi_du[INDI_NUM_ACT];
float indi_du_prev[INDI_NUM_ACT];
float g2_times_du;

// variables needed for estimation
float g1g2_trans_mult[INDI_OUTPUTS][INDI_OUTPUTS];
float g1g2inv[INDI_OUTPUTS][INDI_OUTPUTS];
float actuator_state_filt_vectd[INDI_NUM_ACT];
float actuator_state_filt_vectdd[INDI_NUM_ACT];
float estimation_rate_d[INDI_NUM_ACT];
float estimation_rate_dd[INDI_NUM_ACT];
float du_estimation[INDI_NUM_ACT];
float ddu_estimation[INDI_NUM_ACT];

// The learning rate per axis (roll, pitch, yaw, thrust)
float mu1[INDI_OUTPUTS] = {0.00001, 0.00001, 0.000003, 0.000002};
// The learning rate for the propeller inertia (scaled by 512 wrt mu1)
float mu2 = 0.002;

// other variables
float act_obs[INDI_NUM_ACT];

// Number of actuators used to provide thrust
int32_t num_thrusters;

struct Int32Eulers stab_att_sp_euler;
struct Int32Quat   stab_att_sp_quat;

abi_event rpm_ev;
static void rpm_cb(uint8_t sender_id, uint16_t *rpm, uint8_t num_act);

abi_event thrust_ev;
static void thrust_cb(uint8_t sender_id, float thrust_increment);
float indi_thrust_increment;
bool indi_thrust_increment_set = false;
```

```
188
189 float g1g2_pseudo_inv[INDI_NUM_ACT][INDI_OUTPUTS];
190 float g2[INDI_NUM_ACT] = STABILIZATION_INDI_G2; //scaled by INDI_G_SCALING
191 float g1[INDI_OUTPUTS][INDI_NUM_ACT] = {STABILIZATION_INDI_G1_ROLL,
192                                         STABILIZATION_INDI_G1_PITCH, STABILIZATION_INDI_G1_YAW,
        STABILIZATION_INDI_G1_THRUST
193                                        };
194 float g1g2[INDI_OUTPUTS][INDI_NUM_ACT];
195 float g1_est[INDI_OUTPUTS][INDI_NUM_ACT];
196 float g2_est[INDI_NUM_ACT];
197 float g1_scaled[INDI_OUTPUTS][INDI_NUM_ACT];
198 float g2_scaled[INDI_NUM_ACT];
199 float g1_init[INDI_OUTPUTS][INDI_NUM_ACT];
200 float g2_init[INDI_NUM_ACT];
201
202 Butterworth2LowPass actuator_lowpass_filters[INDI_NUM_ACT];
203 Butterworth2LowPass estimation_input_lowpass_filters[INDI_NUM_ACT];
204 Butterworth2LowPass measurement_lowpass_filters[3];
205 Butterworth2LowPass estimation_output_lowpass_filters[3];
206 Butterworth2LowPass acceleration_lowpass_filter;
207
208 struct FloatVect3 body_accel_f;
209 uint8_t max_iter_inca = 0;
210
211 void init_filters(void);
212
213 #if PERIODIC_TELEMETRY
214 #include "subsystems/datalink/telemetry.h"
215 static void send_indi_g(struct transport_tx *trans, struct link_device *dev)
216 {
217   pprz_msg_send_INDI_G(trans, dev, AC_ID, INDI_NUM_ACT, g1_est[0],
218                        INDI_NUM_ACT, g1_est[1],
219                        INDI_NUM_ACT, g1_est[2],
220                        INDI_NUM_ACT, g1_est[3],
221                        INDI_NUM_ACT, g2_est);
222 }
223
224 static void send_inca(struct transport_tx *trans, struct link_device *dev)
225 {
226   pprz_msg_send_INCA(trans, dev, AC_ID, INDI_OUTPUTS, indi_v,
227                      INDI_NUM_ACT, indi_du,
228                      INDI_NUM_ACT, indi_u,
229                      &max_iter_inca);
230   max_iter_inca = 0;
231 }
232
233 static void send_v_body(struct transport_tx *trans, struct link_device *dev)
234 {
235   float v_body[3] = {speed_body.x, speed_body.y, speed_body.z};
236   pprz_msg_send_V_BODY(trans, dev, AC_ID, 3, v_body);
237 }
238
239 static void send_ahrs_ref_quat(struct transport_tx *trans, struct link_device *dev)
240 {
241   struct Int32Quat *quat = stateGetNedToBodyQuat_i();
242   pprz_msg_send_AHRS_REF_QUAT(trans, dev, AC_ID,
243                               &stab_att_sp_quat.qi,
244                               &stab_att_sp_quat.qx,
245                               &stab_att_sp_quat.qy,
246                               &stab_att_sp_quat.qz,
247                               &(quat->qi),
248                               &(quat->qx),
249                               &(quat->qy),
250                               &(quat->qz));
251 }
252 #endif
253
254 /**
255  * Function that initializes important values upon engaging INDI
256  */
257 void stabilization_indi_init(void)
```

```
258 {
259   // Initialize filters
260   init_filters();
261 
262   AbiBindMsgRPM(RPM_SENSOR_ID, &rpm_ev, rpm_cb);
263   AbiBindMsgTHRUST(THRUST_INCREMENT_ID, &thrust_ev, thrust_cb);
264 
265   float_vect_zero(actuator_state_filt_vectd, INDI_NUM_ACT);
266   float_vect_zero(actuator_state_filt_vectdd, INDI_NUM_ACT);
267   float_vect_zero(estimation_rate_d, INDI_NUM_ACT);
268   float_vect_zero(estimation_rate_dd, INDI_NUM_ACT);
269   float_vect_zero(actuator_state_filt_vect, INDI_NUM_ACT);
270   float_vect_zero(indi_du_prev, INDI_NUM_ACT);
271 
272   //Calculate G1G2_PSEUDO_INVERSE
273   calc_g1g2_pseudo_inv();
274 
275   // Initialize the array of pointers to the rows of g1g2
276   uint8_t i;
277   for (i = 0; i < INDI_OUTPUTS; i++) {
278     Bwls[i] = g1g2[i];
279   }
280 
281   // Remember the initial matrices
282   float_vect_copy(g1_init[0], g1[0], INDI_OUTPUTS * INDI_NUM_ACT);
283   float_vect_copy(g2_init, g2, INDI_NUM_ACT);
284 
285   // Initialize the estimator matrices
286   float_vect_copy(g1_est[0], g1[0], INDI_OUTPUTS * INDI_NUM_ACT);
287   float_vect_copy(g2_est, g2, INDI_NUM_ACT);
288 
289   // Initialize the scaled matrices
290   float_vect_copy(g1_scaled[0], g1[0], INDI_OUTPUTS * INDI_NUM_ACT);
291   float_vect_copy(g2_scaled, g2, INDI_NUM_ACT);
292   scale_surface_effectiveness();
293 
294   // Assume all non-servos are delivering thrust
295   num_thrusters = 0;
296   for (i = 0; i < INDI_NUM_ACT; i++) {
297     num_thrusters += act_is_thrust[i];
298   }
299 
300   // Calculate actuator alpha from first order time constant
301   for (i = 0; i < INDI_NUM_ACT; i++) {
302     act_dyn[i] = 1 - exp(-act_dyn_tau[i] / (PERIODIC_FREQUENCY / run_nth_cycle));
303   }
304 
305 #if PERIODIC_TELEMETRY
306 //  register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_INDI_G, send_indi_g);
307   register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_INCA, send_inca);
308   register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_V_BODY, send_v_body);
309   register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_INDI_G, send_indi_g);
310   register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_AHRS_REF_QUAT, send_ahrs_ref_quat);
311 #endif
312 }
313 
314 /**
315  * Function that resets important values upon engaging INDI.
316  *
317  * Don't reset inputs and filters, because it is unlikely to switch stabilization in flight,
318  * and there are multiple modes that use (the same) stabilization. Resetting the controller
319  * is not so nice when you are flying.
320  * FIXME: Ideally we should detect when coming from something that is not INDI
321  */
322 void stabilization_indi_enter(void)
323 {
324   /* reset psi setpoint to current psi angle */
325   stab_att_sp_euler.psi = stabilization_attitude_get_heading_i();
326 
327   float_vect_zero(du_estimation, INDI_NUM_ACT);
328   float_vect_zero(ddu_estimation, INDI_NUM_ACT);
```

```
329 }
330
331 /**
332  * Function that resets the filters to zeros
333  */
334 void init_filters(void)
335 {
336   // tau = 1/(2*pi*Fc)
337   float tau = 1.0 / (2.0 * M_PI * STABILIZATION_INDI_FILT_CUTOFF);
338   float tau_est = 1.0 / (2.0 * M_PI * STABILIZATION_INDI_ESTIMATION_FILT_CUTOFF);
339   float sample_time = 1.0 / PERIODIC_FREQUENCY;
340   // Filtering of the gyroscope
341   int8_t i;
342   for (i = 0; i < 3; i++) {
343     init_butterworth_2_low_pass(&measurement_lowpass_filters[i], tau, sample_time, 0.0);
344     init_butterworth_2_low_pass(&estimation_output_lowpass_filters[i], tau_est, sample_time, 0.0);
345   }
346
347   // Filtering of the actuators
348   for (i = 0; i < INDI_NUM_ACT; i++) {
349     init_butterworth_2_low_pass(&actuator_lowpass_filters[i], tau, sample_time, 0.0);
350     init_butterworth_2_low_pass(&estimation_input_lowpass_filters[i], tau_est, sample_time, 0.0);
351   }
352
353   // Filtering of the accel body z
354   init_butterworth_2_low_pass(&acceleration_lowpass_filter, tau_est, sample_time, 0.0);
355 }
356
357 /**
358  * Function that calculates the failsafe setpoint
359  */
360 void stabilization_indi_set_failsafe_setpoint(void)
361 {
362   /* set failsafe to zero roll/pitch and current heading */
363   int32_t heading2 = stabilization_attitude_get_heading_i() / 2;
364   PPRZ_ITRIG_COS(stab_att_sp_quat.qi, heading2);
365   stab_att_sp_quat.qx = 0;
366   stab_att_sp_quat.qy = 0;
367   PPRZ_ITRIG_SIN(stab_att_sp_quat.qz, heading2);
368 }
369
370 /**
371  * @param rpy rpy from which to calculate quaternion setpoint
372  *
373  * Function that calculates the setpoint quaternion from rpy
374  */
375 void stabilization_indi_set_rpy_setpoint_i(struct Int32Eulers *rpy)
376 {
377   // stab_att_sp_euler.psi still used in ref..
378   stab_att_sp_euler = *rpy;
379
380   int32_quat_of_eulers(&stab_att_sp_quat, &stab_att_sp_euler);
381 }
382
383 /**
384  * @param cmd 2D command in North East axes
385  * @param heading Heading of the setpoint
386  *
387  * Function that calculates the setpoint quaternion from a command in earth axes
388  */
389 void stabilization_indi_set_earth_cmd_i(struct Int32Vect2 *cmd, int32_t heading)
390 {
391   // stab_att_sp_euler.psi still used in ref..
392   stab_att_sp_euler.psi = heading;
393
394   // compute sp_euler phi/theta for debugging/telemetry
395   /* Rotate horizontal commands to body frame by psi */
396   int32_t psi = stateGetNedToBodyEulers_i()->psi;
397   int32_t s_psi, c_psi;
398   PPRZ_ITRIG_SIN(s_psi, psi);
399   PPRZ_ITRIG_COS(c_psi, psi);
```

```
400   stab_att_sp_euler.phi = (-s_psi * cmd->x + c_psi * cmd->y) >> INT32_TRIG_FRAC;
401   stab_att_sp_euler.theta = -(c_psi * cmd->x + s_psi * cmd->y) >> INT32_TRIG_FRAC;
402
403   quat_from_earth_cmd_i(&stab_att_sp_quat, cmd, heading);
404 }
405
406 /**
407  * @param att_err attitude error
408  * @param rate_control boolean that states if we are in rate control or attitude control
409  * @param in_flight boolean that states if the UAV is in flight or not
410  *
411  * Function that calculates the INDI commands
412  */
413 static void stabilization_indi_calc_cmd(struct Int32Quat *att_err, bool rate_control, bool in_flight)
414 {
415
416   struct FloatRates rate_ref;
417   if (rate_control) { //Check if we are running the rate controller
418     rate_ref.p = (float)radio_control.values[RADIO_ROLL]  / MAX_PPRZ * STABILIZATION_INDI_MAX_RATE;
419     rate_ref.q = (float)radio_control.values[RADIO_PITCH] / MAX_PPRZ * STABILIZATION_INDI_MAX_RATE;
420     rate_ref.r = (float)radio_control.values[RADIO_YAW]   / MAX_PPRZ * STABILIZATION_INDI_MAX_RATE;
421   } else {
422     //calculate the virtual control (reference acceleration) based on a PD controller
423     rate_ref.p = reference_acceleration.err_p * QUAT1_FLOAT_OF_BFP(att_err->qx)
424                  / reference_acceleration.rate_p;
425     rate_ref.q = reference_acceleration.err_q * QUAT1_FLOAT_OF_BFP(att_err->qy)
426                  / reference_acceleration.rate_q;
427     rate_ref.r = reference_acceleration.err_r * QUAT1_FLOAT_OF_BFP(att_err->qz)
428                  / reference_acceleration.rate_r;
429
430     // Possibly we can use some bounding here
431     /*BoundAbs(rate_ref.r, 5.0);*/
432   }
433
434   struct FloatRates *body_rates = stateGetBodyRates_f();
435
436   //calculate the virtual control (reference acceleration) based on a PD controller
437   angular_accel_ref.p = (rate_ref.p - body_rates->p) * reference_acceleration.rate_p;
438   angular_accel_ref.q = (rate_ref.q - body_rates->q) * reference_acceleration.rate_q;
439   angular_accel_ref.r = (rate_ref.r - body_rates->r) * reference_acceleration.rate_r;
440
441   g2_times_du = 0.0;
442   int8_t i;
443   for (i = 0; i < INDI_NUM_ACT; i++) {
444     g2_times_du += g2[i] * indi_du[i];
445   }
446   //G2 is scaled by INDI_G_SCALING to make it readable
447   g2_times_du = g2_times_du / INDI_G_SCALING;
448
449   float v_thrust = 0.0;
450   if (indi_thrust_increment_set && in_flight) {
451     v_thrust = indi_thrust_increment;
452
453     //update thrust command such that the current is correctly estimated
454     stabilization_cmd[COMMAND_THRUST] = 0;
455     for (i = 0; i < INDI_NUM_ACT; i++) {
456       stabilization_cmd[COMMAND_THRUST] += actuator_state[i] * (float) act_is_thrust[i];
457     }
458     stabilization_cmd[COMMAND_THRUST] /= (float) num_thrusters;
459
460   } else {
461     // incremental thrust
462     for (i = 0; i < INDI_NUM_ACT; i++) {
463       v_thrust +=
464         (stabilization_cmd[COMMAND_THRUST] - actuator_state_filt_vect[i]) * Bwls[3][i];
465     }
466   }
467
468   // The control objective in array format
469   indi_v[0] = (angular_accel_ref.p - angular_acceleration[0]);
470   indi_v[1] = (angular_accel_ref.q - angular_acceleration[1]);
```

```
471   indi_v[2] = (angular_accel_ref.r - angular_acceleration[2] + g2_times_du);
472   indi_v[3] = v_thrust;
473
474   // The control objective in array format
475   indi_v_abs[0] = angular_accel_ref.p;
476   indi_v_abs[1] = angular_accel_ref.q;
477   indi_v_abs[2] = angular_accel_ref.r;
478   indi_v_abs[3] = v_thrust;
479
480 #if STABILIZATION_INDI_ALLOCATION_PSEUDO_INVERSE
481   // Calculate the increment for each actuator
482   for (i = 0; i < INDI_NUM_ACT; i++) {
483     indi_du[i] = (g1g2_pseudo_inv[i][0] * indi_v[0])
484                + (g1g2_pseudo_inv[i][1] * indi_v[1])
485                + (g1g2_pseudo_inv[i][2] * indi_v[2])
486                + (g1g2_pseudo_inv[i][3] * indi_v[3]);
487   }
488 #else
489   // Calculate the min and max increments
490   for (i = 0; i < INDI_NUM_ACT; i++) {
491     du_min[i] = MAX_PPRZ * act_is_servo[i] - actuator_state_filt_vect[i];
492     du_max[i] = MAX_PPRZ - actuator_state_filt_vect[i];
493     du_pref[i] = act_pref[i] - actuator_state_filt_vect[i];
494   }
495
496   // Scale control surfaces with forward velocity
497   scale_surface_effectiveness();
498
499   // WLS Control Allocator
500   uint8_t iter = wls_alloc(indi_du, indi_v, du_min, du_max, Bwls, indi_du_prev, 0, Wv, Wu, du_pref, 10000,
        10);
501   float_vect_copy(indi_du_prev, indi_du, INDI_NUM_ACT);
502 #endif
503
504   if (iter > max_iter_inca) {
505     max_iter_inca = iter;
506   }
507
508   // Add the increments to the actuators
509   float_vect_sum(indi_u, actuator_state_filt_vect, indi_du, INDI_NUM_ACT);
510
511   // Bound the inputs to the actuators
512   for (i = 0; i < INDI_NUM_ACT; i++) {
513     if (act_is_servo[i]) {
514       BoundAbs(indi_u[i], MAX_PPRZ);
515     } else {
516       Bound(indi_u[i], 0, MAX_PPRZ);
517     }
518   }
519
520   //Don't increment if not flying (not armed)
521   if (!in_flight) {
522     float_vect_zero(indi_u, INDI_NUM_ACT);
523     float_vect_zero(indi_du, INDI_NUM_ACT);
524   }
525
526   // Propagate actuator filters
527   get_actuator_state();
528   for (i = 0; i < INDI_NUM_ACT; i++) {
529     update_butterworth_2_low_pass(&actuator_lowpass_filters[i], actuator_state[i]);
530     update_butterworth_2_low_pass(&estimation_input_lowpass_filters[i], actuator_state[i]);
531     actuator_state_filt_vect[i] = actuator_lowpass_filters[i].o[0];
532
533     // calculate derivatives for estimation
534     float actuator_state_filt_vectd_prev = actuator_state_filt_vectd[i];
535     actuator_state_filt_vectd[i] = (estimation_input_lowpass_filters[i].o[0] -
        estimation_input_lowpass_filters[i].o[1]) * PERIODIC_FREQUENCY;
536     actuator_state_filt_vectdd[i] = (actuator_state_filt_vectd[i] - actuator_state_filt_vectd_prev) *
        PERIODIC_FREQUENCY;
537   }
538
```

```
539   // Use online effectiveness estimation only when flying
540   if (in_flight && indi_use_adaptive) {
541     lms_estimation();
542   }
543
544   /*Commit the actuator command*/
545   for (i = 0; i < INDI_NUM_ACT; i++) {
546     actuators_pprz[i] = (int16_t) indi_u[i];
547   }
548
549 }
550
551 /**
552  * @param enable_integrator
553  * @param rate_control boolean that determines if we are in rate control or attitude control
554  *
555  * Function that should be called to run the INDI controller
556  */
557 void stabilization_indi_run(bool in_flight, bool rate_control)
558 {
559
560   /* compute the INDI command once every run_nth_cycle cycles*/
561   if (num_cycle < run_nth_cycle) {
562     num_cycle++;
563     return;
564   }
565
566   /* Propagate the filter on the gyroscopes */
567   struct FloatRates *body_rates = stateGetBodyRates_f();
568   float rate_vect[3] = {body_rates->p, body_rates->q, body_rates->r};
569   int8_t i;
570   for (i = 0; i < 3; i++) {
571     update_butterworth_2_low_pass(&measurement_lowpass_filters[i], rate_vect[i]);
572     update_butterworth_2_low_pass(&estimation_output_lowpass_filters[i], rate_vect[i]);
573
574     //Calculate the angular acceleration via finite difference
575     angular_acceleration[i] = (measurement_lowpass_filters[i].o[0]
576                                - measurement_lowpass_filters[i].o[1]) * PERIODIC_FREQUENCY;
577
578     // Calculate derivatives for estimation
579     float estimation_rate_d_prev = estimation_rate_d[i];
580     estimation_rate_d[i] = (estimation_output_lowpass_filters[i].o[0] - estimation_output_lowpass_filters[
      i].o[1]) * PERIODIC_FREQUENCY;
581     estimation_rate_dd[i] = (estimation_rate_d[i] - estimation_rate_d_prev) * PERIODIC_FREQUENCY;
582   }
583
584   /* attitude error                          */
585   struct Int32Quat att_err;
586   struct Int32Quat *att_quat = stateGetNedToBodyQuat_i();
587   int32_quat_inv_comp(&att_err, att_quat, &stab_att_sp_quat);
588   /* wrap it in the shortest direction       */
589   int32_quat_wrap_shortest(&att_err);
590   int32_quat_normalize(&att_err);
591
592   /* compute the INDI command */
593   stabilization_indi_calc_cmd(&att_err, rate_control, in_flight);
594
595   // Set the stab_cmd to 42 to indicate that it is not used
596   stabilization_cmd[COMMAND_ROLL] = 42;
597   stabilization_cmd[COMMAND_PITCH] = 42;
598   stabilization_cmd[COMMAND_YAW] = 42;
599
600   // Reset thrust increment boolean
601   indi_thrust_increment_set = false;
602
603   // Reset cycle number
604   num_cycle = 1;
605
606 }
607
608 // This function reads rc commands
```

```
609 void stabilization_indi_read_rc(bool in_flight, bool in_carefree, bool coordinated_turn)
610 {
611   struct FloatQuat q_sp;
612 #if USE_EARTH_BOUND_RC_SETPOINT
613   stabilization_attitude_read_rc_setpoint_quat_earth_bound_f(&q_sp, in_flight, in_carefree,
      coordinated_turn);
614 #else
615   stabilization_attitude_read_rc_setpoint_quat_f(&q_sp, in_flight, in_carefree, coordinated_turn);
616 #endif
617
618   QUAT_BFP_OF_REAL(stab_att_sp_quat, q_sp);
619 }
620
621 /**
622  * Function that tries to get actuator feedback.
623  *
624  * If this is not available it will use a first order filter to approximate the actuator state.
625  * It is also possible to model rate limits (unit: PPRZ/loop cycle)
626  */
627 void get_actuator_state(void)
628 {
629 #if INDI_RPM_FEEDBACK
630   float_vect_copy(actuator_state, act_obs, INDI_NUM_ACT);
631 #else
632   //actuator dynamics
633   int8_t i;
634   float UNUSED prev_actuator_state;
635   for (i = 0; i < INDI_NUM_ACT; i++) {
636     prev_actuator_state = actuator_state[i];
637
638     actuator_state[i] = actuator_state[i]
639                         + act_dyn[i] * (indi_u[i] - actuator_state[i]);
640
641 #ifdef STABILIZATION_INDI_ACT_RATE_LIMIT
642     if ((actuator_state[i] - prev_actuator_state) > act_rate_limit[i]) {
643       actuator_state[i] = prev_actuator_state + act_rate_limit[i];
644     } else if ((actuator_state[i] - prev_actuator_state) < -act_rate_limit[i]) {
645       actuator_state[i] = prev_actuator_state - act_rate_limit[i];
646     }
647 #endif
648   }
649
650 #endif
651 }
652
653 /**
654  * @param ddx_error error in output change
655  * @param i row of the matrix element
656  * @param j column of the matrix element
657  * @param mu learning rate
658  *
659  * Function that calculates an element of the G1 matrix.
660  * The elements are stored in a different matrix,
661  * because the old matrix is necessary to caclulate more elements.
662  */
663 void calc_g1_element(float ddx_error, int8_t i, int8_t j, float mu)
664 {
665   g1_est[i][j] = g1_est[i][j] - du_estimation[j] * mu * ddx_error;
666 }
667
668 /**
669  * @param ddx_error error in output change
670  * @param j column of the matrix element
671  * @param mu learning rate
672  *
673  * Function that calculates an element of the G2 matrix.
674  * The elements are stored in a different matrix,
675  * because the old matrix is necessary to caclulate more elements.
676  */
677 void calc_g2_element(float ddx_error, int8_t j, float mu)
678 {
```

```
679   g2_est[j] = g2_est[j] − ddu_estimation[j] * mu * ddx_error;
680 }
681
682 /**
683  * Function that estimates the control effectiveness of each actuator online.
684  * It is assumed that disturbances do not play a large role.
685  * All elements of the G1 and G2 matrices are be estimated.
686  */
687 void lms_estimation(void)
688 {
689
690   // Get the acceleration in body axes
691   struct Int32Vect3 *body_accel_i;
692   body_accel_i = stateGetAccelBody_i();
693   ACCELS_FLOAT_OF_BFP(body_accel_f, *body_accel_i);
694
695   // Filter the acceleration in z axis
696   update_butterworth_2_low_pass(&acceleration_lowpass_filter, body_accel_f.z);
697
698   // Calculate the derivative of the acceleration via finite difference
699   float indi_accel_d = (acceleration_lowpass_filter.o[0]
700                         − acceleration_lowpass_filter.o[1]) * PERIODIC_FREQUENCY;
701
702   // scale the inputs to avoid numerical errors
703   float_vect_smul(du_estimation, actuator_state_filt_vectd, 0.001, INDI_NUM_ACT);
704   float_vect_smul(ddu_estimation, actuator_state_filt_vectdd, 0.001 / PERIODIC_FREQUENCY, INDI_NUM_ACT);
705
706   float ddx_estimation[INDI_OUTPUTS] = {estimation_rate_dd[0], estimation_rate_dd[1], estimation_rate_dd
        [2], indi_accel_d};
707
708   //Estimation of G
709   // TODO: only estimate when du_norm2 is large enough (enough input)
710   /*float du_norm2 = du_estimation[0]*du_estimation[0] + du_estimation[1]*du_estimation[1] +du_estimation
        [2]*du_estimation[2] + du_estimation[3]*du_estimation[3];*/
711   int8_t i;
712   for (i = 0; i < INDI_OUTPUTS; i++) {
713     // Calculate the error between prediction and measurement
714     float ddx_error = − ddx_estimation[i];
715     int8_t j;
716     for (j = 0; j < INDI_NUM_ACT; j++) {
717       ddx_error += g1_est[i][j] * du_estimation[j];
718       if (i == 2) {
719         // Changing the momentum of the rotors gives a counter torque
720         ddx_error += g2_est[j] * ddu_estimation[j];
721       }
722     }
723
724     // when doing the yaw axis, also use G2
725     if (i == 2) {
726       for (j = 0; j < INDI_NUM_ACT; j++) {
727         calc_g2_element(ddx_error, j, mu2);
728       }
729     } else if (i == 3) {
730       // If the acceleration change is very large (rough landing), don't adapt
731       if (fabs(indi_accel_d) > 60.0) {
732         ddx_error = 0.0;
733       }
734     }
735
736     // Calculate the row of the G1 matrix corresponding to this axis
737     for (j = 0; j < INDI_NUM_ACT; j++) {
738       calc_g1_element(ddx_error, i, j, mu1[i]);
739     }
740   }
741
742   bound_g_mat();
743
744   // Save the calculated matrix to G1 and G2
745   // until thrust is included, first part of the array
746   float_vect_copy(g1[0], g1_est[0], INDI_OUTPUTS * INDI_NUM_ACT);
747   float_vect_copy(g2, g2_est, INDI_NUM_ACT);
```

```
748
749 #if STABILIZATION_INDI_ALLOCATION_PSEUDO_INVERSE
750   // Calculate the inverse of (G1+G2)
751   calc_g1g2_pseudo_inv();
752 #endif
753 }
754
755 /**
756  * Function that calculates the pseudo-inverse of (G1+G2).
757  */
758 void calc_g1g2_pseudo_inv(void)
759 {
760
761   //sum of G1 and G2
762   int8_t i;
763   int8_t j;
764   for (i = 0; i < INDI_OUTPUTS; i++) {
765     for (j = 0; j < INDI_NUM_ACT; j++) {
766       if (i != 2) {
767         g1g2[i][j] = g1[i][j] / INDI_G_SCALING;
768       } else {
769         g1g2[i][j] = (g1[i][j] + g2[j]) / INDI_G_SCALING;
770       }
771     }
772   }
773
774   //G1G2*transpose(G1G2)
775   //calculate matrix multiplication of its transpose INDI_OUTPUTSxnum_act x num_actxINDI_OUTPUTS
776   float element = 0;
777   int8_t row;
778   int8_t col;
779   for (row = 0; row < INDI_OUTPUTS; row++) {
780     for (col = 0; col < INDI_OUTPUTS; col++) {
781       element = 0;
782       for (i = 0; i < INDI_NUM_ACT; i++) {
783         element = element + g1g2[row][i] * g1g2[col][i];
784       }
785       g1g2_trans_mult[row][col] = element;
786     }
787   }
788
789   //there are numerical errors if the scaling is not right.
790   float_vect_scale(g1g2_trans_mult[0], 100.0, INDI_OUTPUTS * INDI_OUTPUTS);
791
792   //inverse of 4x4 matrix
793   float_mat_inv_4d(g1g2inv[0], g1g2_trans_mult[0]);
794
795   //scale back
796   float_vect_scale(g1g2inv[0], 100.0, INDI_OUTPUTS * INDI_OUTPUTS);
797
798   //G1G2'*G1G2inv
799   //calculate matrix multiplication INDI_NUM_ACTxINDI_OUTPUTS x INDI_OUTPUTSxINDI_OUTPUTS
800   for (row = 0; row < INDI_NUM_ACT; row++) {
801     for (col = 0; col < INDI_OUTPUTS; col++) {
802       element = 0;
803       for (i = 0; i < INDI_OUTPUTS; i++) {
804         element = element + g1g2[i][row] * g1g2inv[col][i];
805       }
806       g1g2_pseudo_inv[row][col] = element;
807     }
808   }
809 }
810
811 static void rpm_cb(uint8_t __attribute__((unused)) sender_id, uint16_t UNUSED *rpm, uint8_t UNUSED num_act
      )
812 {
813 #if INDI_RPM_FEEDBACK
814   #ifndef ACTUATORS_PWM_H
815     int8_t i;
816     for (i = 0; i < num_act; i++) {
817       act_obs[i] = (rpm[i] - get_servo_min(i));
```

```
818       act_obs[i] *= (MAX_PPRZ / (float)(get_servo_max(i) - get_servo_min(i)));
819       Bound(act_obs[i], 0, MAX_PPRZ);
820     }
821   #else
822     int8_t i;
823     for (i = 0; i < num_act; i++) {
824       act_obs[i] = (rpm[i] - get_servo_min_PWM(i));
825       act_obs[i] *= (MAX_PPRZ / (float)(get_servo_max_PWM(i) - get_servo_min_PWM(i)));
826       Bound(act_obs[i], 0, MAX_PPRZ);
827     }
828   #endif
829 #endif
830 }
831 
832 /**
833  * ABI callback that obtains the thrust increment from guidance INDI
834  */
835 static void thrust_cb(uint8_t UNUSED sender_id, float thrust_increment)
836 {
837   indi_thrust_increment = thrust_increment;
838   indi_thrust_increment_set = true;
839 }
840 
841 static void bound_g_mat(void)
842 {
843   int8_t i;
844   int8_t j;
845   for (j = 0; j < INDI_NUM_ACT; j++) {
846     float max_limit;
847     float min_limit;
848 
849     // Limit the values of the estimated G1 matrix
850     for (i = 0; i < INDI_OUTPUTS; i++) {
851       if (g1_init[i][j] > 0.0) {
852         max_limit = g1_init[i][j] * INDI_ALLOWED_G_FACTOR;
853         min_limit = g1_init[i][j] / INDI_ALLOWED_G_FACTOR;
854       } else {
855         max_limit = g1_init[i][j] / INDI_ALLOWED_G_FACTOR;
856         min_limit = g1_init[i][j] * INDI_ALLOWED_G_FACTOR;
857       }
858 
859       if (g1_est[i][j] > max_limit) {
860         g1_est[i][j] = max_limit;
861       }
862       if (g1_est[i][j] < min_limit) {
863         g1_est[i][j] = min_limit;
864       }
865     }
866 
867     // Do the same for the G2 matrix
868     if (g2_init[j] > 0.0) {
869       max_limit = g2_init[j] * INDI_ALLOWED_G_FACTOR;
870       min_limit = g2_init[j] / INDI_ALLOWED_G_FACTOR;
871     } else {
872       max_limit = g2_init[j] / INDI_ALLOWED_G_FACTOR;
873       min_limit = g2_init[j] * INDI_ALLOWED_G_FACTOR;
874     }
875 
876     if (g2_est[j] > max_limit) {
877       g2_est[j] = max_limit;
878     }
879     if (g2_est[j] < min_limit) {
880       g2_est[j] = min_limit;
881     }
882   }
883 }
884 
885 /**
886  * Function that scales the control effectiveness of the control surfaces
887  */
888 void scale_surface_effectiveness(void)
```

```
889 {
890
891   // Get body velocity velocity (ideally airspeed velocity)
892   struct FloatRMat *ned_to_body_rmat = stateGetNedToBodyRMat_f();
893   struct NedCoor_f *ned_speed_f = stateGetSpeedNed_f();
894   struct FloatVect3 speed_ned = {ned_speed_f->x, ned_speed_f->y, ned_speed_f->z};
895   float_rmat_vmult(&speed_body, ned_to_body_rmat, &speed_ned);
896
897   // Bound measured forward speed for only large enough positive values
898   if (speed_body.x < 0.1) {
899     speed_body.x = 0;
900   }
901
902   // Iterate over actuators
903   int8_t j;
904   for (j = 0; j < INDI_NUM_ACT; j++) {
905
906   // If actuator is control surface, scale with forward speed
907     if (act_is_surface[j]) {
908       int8_t i;
909       for (i = 0; i < INDI_OUTPUTS; i++) {
910     g1_scaled[i][j] = g1_init[i][j] * speed_body.x * speed_body.x;
911     }
912     g2_scaled[j] = g2_init[j] * speed_body.x * speed_body.x;
913   }
914
915   // If actuator is not a control surface, don't scale
916   else {
917       int8_t i;
918       for (i = 0; i < INDI_OUTPUTS; i++) {
919     g1_scaled[i][j] = g1_init[i][j];
920     }
921     g2_scaled[j] = g2_init[j];
922   }
923
924   }
925
926   // Save the calculated matrix to G1 and G2
927   float_vect_copy(g1[0], g1_scaled[0], INDI_OUTPUTS * INDI_NUM_ACT);
928   float_vect_copy(g2, g2_scaled, INDI_NUM_ACT);
929
930   //Recalculate G1G2_PSEUDO_INVERSE
931   calc_g1g2_pseudo_inv();
932
933   // Update the array of pointers to the rows of g1g2
934   uint8_t i;
935   for (i = 0; i < INDI_OUTPUTS; i++) {
936     Bwls[i] = g1g2[i];
937   }
938
939 }
```

D

Source code of the XINCA module

This code is based on an existing Paparazzi UAV outer loop INDI module by Smeur et al. [1, 2].

[1] Ewoud J.J. Smeur, Qiping Chu, and Guido C.H.E. De Croon. Cascaded incremental nonlinear dynamic inversion control for mav disturbance rejection. 2018. doi: https://doi.org/10.1016/j.conengprac.2018.01.003.

[2] Ewoud J.J. Smeur, Qiping Chu, and Guido C.H.E. De Croon. Gust disturbance alleviation with incremental nonlinear dynamic inversion. 2016. doi: https://doi.org/10.1109/IROS.2016.7759827.

```
/*
 * Original module: Copyright (C) 2015 Ewoud Smeur <ewoud.smeur@gmail.com>
 * XINCA extension: Copyright (C) 2020 Jan Karssies <hjkarssies@gmail.com>
 *
 * This file is part of paparazzi.
 *
 * paparazzi is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * paparazzi is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with paparazzi; see the file COPYING. If not, write to
 * the Free Software Foundation, 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/**
 * @file firmwares/rotorcraft/guidance/guidance_indi.c
 *
 * A guidance mode based on Extended Incremental Nonlinear Control Allocation
 *
 * Based on the papers:
 * Cascaded Incremental Nonlinear Dynamic Inversion Control for MAV Disturbance Rejection
 * https://www.researchgate.net/publication/312907985
   _Cascaded_Incremental_Nonlinear_Dynamic_Inversion_Control_for_MAV_Disturbance_Rejection
 *
 * Gust Disturbance Alleviation with Incremental Nonlinear Dynamic Inversion
 * https://www.researchgate.net/publication/309212603
   _Gust_Disturbance_Alleviation_with_Incremental_Nonlinear_Dynamic_Inversion
 *
 * Extended Nonlinear Control Allocation on the TU Delft Quadplane - H.J. Karssies
 * http://repository.tudelft.nl/
 */
#include "generated/airframe.h"
```

```
39 #include "firmwares/rotorcraft/guidance/guidance_xinca.h"
40 #include "subsystems/ins/ins_int.h"
41 #include "subsystems/radio_control.h"
42 #include "subsystems/actuators.h"
43 #include "state.h"
44 #include "subsystems/imu.h"
45 #include "firmwares/rotorcraft/guidance/guidance_h.h"
46 #include "firmwares/rotorcraft/guidance/guidance_v.h"
47 #include "firmwares/rotorcraft/stabilization/stabilization_attitude.h"
48 #include "firmwares/rotorcraft/autopilot_rc_helpers.h"
49 #include "mcu_periph/sys_time.h"
50 #include "autopilot.h"
51 #include "stabilization/stabilization_attitude_ref_quat_int.h"
52 #include "firmwares/rotorcraft/stabilization.h"
53 #include "filters/low_pass_filter.h"
54 #include "subsystems/abi.h"
55 #include "firmwares/rotorcraft/guidance/wls/wls_alloc_guidance.h"
56
57 // The acceleration reference is calculated with these gains. If you use GPS,
58 // they are probably limited by the update rate of your GPS. The default
59 // values are tuned for 4 Hz GPS updates. If you have high speed position updates, the
60 // gains can be higher, depending on the speed of the inner loop.
61 #ifdef GUIDANCE_INDI_POS_GAIN
62 float guidance_indi_pos_gain = GUIDANCE_INDI_POS_GAIN;
63 #else
64 float guidance_indi_pos_gain = 0.5;
65 #endif
66
67 #ifdef GUIDANCE_INDI_SPEED_GAIN
68 float guidance_indi_speed_gain = GUIDANCE_INDI_SPEED_GAIN;
69 #else
70 float guidance_indi_speed_gain = 1.8;
71 #endif
72
73 #ifndef GUIDANCE_INDI_ACCEL_SP_ID
74 #define GUIDANCE_INDI_ACCEL_SP_ID ABI_BROADCAST
75 #endif
76 abi_event accel_sp_ev;
77 static void accel_sp_cb(uint8_t sender_id, uint8_t flag, struct FloatVect3 *accel_sp);
78 struct FloatVect3 indi_accel_sp = {0.0, 0.0, 0.0};
79 bool indi_accel_sp_set_2d = false;
80 bool indi_accel_sp_set_3d = false;
81
82 struct FloatVect3 sp_accel = {0.0, 0.0, 0.0};
83
84 static void guidance_indi_filter_actuators(void);
85
86 #ifndef GUIDANCE_INDI_FILTER_CUTOFF
87 #ifdef STABILIZATION_INDI_FILT_CUTOFF
88 #define GUIDANCE_INDI_FILTER_CUTOFF STABILIZATION_INDI_FILT_CUTOFF
89 #else
90 #define GUIDANCE_INDI_FILTER_CUTOFF 3.0
91 #endif
92 #endif
93
94 float act_z = 0;
95 float act_x = 0;
96
97 Butterworth2LowPass filt_accel_ned[3];
98 Butterworth2LowPass roll_filt;
99 Butterworth2LowPass pitch_filt;
100 Butterworth2LowPass act_z_filt;
101 Butterworth2LowPass act_x_filt;
102
103 float control_increment[XINCA_NUM_ACT]; // [dtheta, dphi, dact_z, dact_x]
104
105 float filter_cutoff = GUIDANCE_INDI_FILTER_CUTOFF;
106 float guidance_indi_max_bank = GUIDANCE_H_MAX_BANK;
107
108 float time_of_accel_sp_2d = 0.0;
109 float time_of_accel_sp_3d = 0.0;
```

```
110
111 struct FloatEulers guidance_euler_cmd;
112 float act_z_in;
113 float act_x_in;
114
115 //XINCA specific parameters
116 int num_iter_xinca = 0;
117 int num_cycle_xinca = 1; // Cycle count (resets every nth cycle)
118 #ifdef GUIDANCE_XINCA_NTH_CYCLE
119 int run_nth_cycle_xinca = GUIDANCE_XINCA_NTH_CYCLE; // Run every nth cycle
120 #else
121 int run_nth_cycle_xinca = 1; // Run every nth cycle
122 #endif
123
124 float G[XINCA_OUTPUTS][XINCA_NUM_ACT];
125 float *B[XINCA_OUTPUTS];
126 float v_xinca[XINCA_OUTPUTS];
127 float u_xinca[XINCA_NUM_ACT];
128 float du_xinca[XINCA_NUM_ACT];
129 float du_min_xinca[XINCA_NUM_ACT];
130 float du_max_xinca[XINCA_NUM_ACT];
131 float du_pref_xinca[XINCA_NUM_ACT];
132 float du_prev_xinca[XINCA_NUM_ACT];
133
134 float c_l_alpha = GUIDANCE_XINCA_C_L_ALPHA;
135 float c_z_thrust = GUIDANCE_XINCA_C_Z_THRUST;
136 float c_x_tail_rotor = GUIDANCE_XINCA_C_X_TAIL_ROTOR;
137 float mass = GUIDANCE_XINCA_MASS;
138 float wing_surface = GUIDANCE_XINCA_WING_SURFACE;
139
140 #ifdef GUIDANCE_XINCA_RHO
141 float rho = GUIDANCE_XINCA_RHO;
142 #else
143 float rho = 1.225;
144 #endif
145
146 #ifdef GUIDANCE_XINCA_GRAVITY
147 float gravity = GUIDANCE_XINCA_GRAVITY;
148 #else
149 float gravity = 9.81;
150 #endif
151
152 #ifdef GUIDANCE_XINCA_ACT_Z_TAU
153 float act_z_tau = GUIDANCE_XINCA_ACT_Z_TAU;
154 #else
155 float act_z_tau = 30;
156 #endif
157 float act_z_dyn;
158
159 #ifdef GUIDANCE_XINCA_ACT_X_TAU
160 float act_x_tau = GUIDANCE_XINCA_ACT_X_TAU;
161 #else
162 float act_x_tau = 60;
163 #endif
164 float act_x_dyn;
165
166 #ifdef GUIDANCE_XINCA_U_PREF
167 float u_pref[XINCA_NUM_ACT] = GUIDANCE_XINCA_U_PREF;
168 #else
169 float u_pref[XINCA_NUM_ACT] = {0, 0, 0, 0};
170 #endif
171
172 #ifdef GUIDANCE_XINCA_W_ACC
173 float W_acc[XINCA_OUTPUTS] = GUIDANCE_XINCA_W_ACC;
174 #else
175 float W_acc[XINCA_OUTPUTS] = {10, 10, 1};
176 #endif
177
178 #ifdef GUIDANCE_XINCA_W_ACT
179 float W_act[XINCA_NUM_ACT] = GUIDANCE_XINCA_W_ACT;
180 #else
```

```
181 float W_act[XINCA_NUM_ACT] = {10, 10, 100, 1};
182 #endif
183
184 #ifdef GUIDANCE_XINCA_GAMMA
185 float gamma_sq = GUIDANCE_XINCA_GAMMA;
186 #else
187 float gamma_sq = 10000;
188 #endif
189
190 #ifdef GUIDANCE_XINCA_H_THRES
191 float h_thres = GUIDANCE_XINCA_H_THRES;
192 #else
193 float h_thres = 0.2;
194 #endif
195
196 uint8_t max_iter_xinca = 0;
197
198 static void guidance_indi_propagate_filters(struct FloatEulers *eulers);
199 static void guidance_xinca_calcG_yxz(struct FloatEulers *euler_yxz);
200
201 #if PERIODIC_TELEMETRY
202 #include "subsystems/datalink/telemetry.h"
203 static void send_xinca(struct transport_tx *trans, struct link_device *dev)
204 {
205   pprz_msg_send_XINCA(trans, dev, AC_ID, XINCA_OUTPUTS, v_xinca,
206                       XINCA_NUM_ACT, du_xinca,
207                       XINCA_NUM_ACT, u_xinca,
208                       &max_iter_xinca);
209   max_iter_xinca = 0;
210 }
211 #endif
212
213 /**
214  * @brief Init function
215  */
216 void guidance_indi_init(void)
217 {
218   AbiBindMsgACCEL_SP(GUIDANCE_INDI_ACCEL_SP_ID, &accel_sp_ev, accel_sp_cb);
219
220 #if PERIODIC_TELEMETRY
221   register_periodic_telemetry(DefaultPeriodic, PPRZ_MSG_ID_XINCA, send_xinca);
222 #endif
223 }
224
225 /**
226  *
227  * Call upon entering indi guidance
228  */
229 void guidance_indi_enter(void)
230 {
231
232   act_z_dyn = 1 - exp(-act_z_tau / (PERIODIC_FREQUENCY / (float) run_nth_cycle_xinca));
233   act_z_in = 0;
234   act_z = act_z_in;
235
236   act_x_dyn = 1 - exp(-act_x_tau / (PERIODIC_FREQUENCY / (float) run_nth_cycle_xinca));
237   act_x_in = 0;
238   act_x = act_x_in;
239
240   float_vect_zero(du_prev_xinca, XINCA_NUM_ACT);
241
242   float tau = 1.0 / (2.0 * M_PI * filter_cutoff);
243   float sample_time = 1.0 / (PERIODIC_FREQUENCY / (float) run_nth_cycle_xinca);
244   for (int8_t i = 0; i < 3; i++) {
245     init_butterworth_2_low_pass(&filt_accel_ned[i], tau, sample_time, 0.0);
246   }
247
248   init_butterworth_2_low_pass(&roll_filt, tau, sample_time, stateGetNedToBodyEulers_f()->phi);
249   init_butterworth_2_low_pass(&pitch_filt, tau, sample_time, stateGetNedToBodyEulers_f()->theta);
250   init_butterworth_2_low_pass(&act_z_filt, act_z_tau, sample_time, act_z_in);
251   init_butterworth_2_low_pass(&act_x_filt, act_x_tau, sample_time, act_x_in);
```

```
252
253 }
254
255 /**
256  * @param heading_sp the desired heading [rad]
257  *
258  * main indi guidance function
259  */
260 void guidance_indi_run(float *heading_sp)
261 {
262
263   // Only compute the XINCA command once every run_nth_cycle cycles
264   if (num_cycle_xinca >= (float) run_nth_cycle_xinca) {
265     num_cycle_xinca = 1;
266   } else {
267     num_cycle_xinca += 1;
268     return;
269   }
270
271   struct FloatEulers eulers_yxz;
272   struct FloatQuat * statequat = stateGetNedToBodyQuat_f();
273   float_eulers_of_quat_yxz(&eulers_yxz, statequat);
274
275   // Filter accel to get rid of noise and filter attitude to synchronize with accel
276   guidance_indi_propagate_filters(&eulers_yxz);
277
278   // Linear controller to find the acceleration setpoint from position and velocity
279   float pos_x_err = POS_FLOAT_OF_BFP(guidance_h.ref.pos.x) - stateGetPositionNed_f()->x;
280   float pos_y_err = POS_FLOAT_OF_BFP(guidance_h.ref.pos.y) - stateGetPositionNed_f()->y;
281   float pos_z_err = POS_FLOAT_OF_BFP(guidance_v_z_ref - stateGetPositionNed_i()->z);
282
283   float speed_sp_x = pos_x_err * guidance_indi_pos_gain;
284   float speed_sp_y = pos_y_err * guidance_indi_pos_gain;
285   float speed_sp_z = pos_z_err * guidance_indi_pos_gain;
286
287   // If the acceleration setpoint is set over ABI message
288   if (indi_accel_sp_set_2d) {
289     sp_accel.x = indi_accel_sp.x;
290     sp_accel.y = indi_accel_sp.y;
291     // In 2D the vertical motion is derived from the flight plan
292     sp_accel.z = (speed_sp_z - stateGetSpeedNed_f()->z) * guidance_indi_speed_gain;
293     float dt = get_sys_time_float() - time_of_accel_sp_2d;
294     // If the input command is not updated after a timeout, switch back to flight plan control
295     if (dt > 0.5) {
296       indi_accel_sp_set_2d = false;
297     }
298   } else if (indi_accel_sp_set_3d) {
299     sp_accel.x = indi_accel_sp.x;
300     sp_accel.y = indi_accel_sp.y;
301     sp_accel.z = indi_accel_sp.z;
302     float dt = get_sys_time_float() - time_of_accel_sp_3d;
303     // If the input command is not updated after a timeout, switch back to flight plan control
304     if (dt > 0.5) {
305       indi_accel_sp_set_3d = false;
306     }
307   } else {
308     sp_accel.x = (speed_sp_x - stateGetSpeedNed_f()->x) * guidance_indi_speed_gain;
309     sp_accel.y = (speed_sp_y - stateGetSpeedNed_f()->y) * guidance_indi_speed_gain;
310     sp_accel.z = (speed_sp_z - stateGetSpeedNed_f()->z) * guidance_indi_speed_gain;
311   }
312
313 #if GUIDANCE_INDI_RC_DEBUG
314 #warning "GUIDANCE_INDI_RC_DEBUG lets you control the accelerations via RC, but disables autonomous flight
      !"
315   //for rc control horizontal, rotate from body axes to NED
316   float psi = stateGetNedToBodyEulers_f()->psi;
317   float rc_x = -(radio_control.values[RADIO_PITCH] / 9600.0) * 8.0;
318   float rc_y = (radio_control.values[RADIO_ROLL] / 9600.0) * 8.0;
319   sp_accel.x = cosf(psi) * rc_x - sinf(psi) * rc_y;
320   sp_accel.y = sinf(psi) * rc_x + cosf(psi) * rc_y;
321
```

```
322   //for rc vertical control
323   sp_accel.z = -(radio_control.values[RADIO_THROTTLE] - 4500) * 8.0 / 9600.0;
324 #endif
325
326   // Calculate matrix of partial derivatives
327   guidance_xinca_calcG_yxz(&eulers_yxz);
328
329   struct FloatVect3 a_diff = { sp_accel.x - filt_accel_ned[0].o[0],sp_accel.y - filt_accel_ned[1].o[0],
        sp_accel.z - filt_accel_ned[2].o[0]};
330
331   // Bound the acceleration error so that the linearization still holds
332   Bound(a_diff.x, -6.0, 6.0);
333   Bound(a_diff.y, -6.0, 6.0);
334   Bound(a_diff.z, -9.0, 9.0);
335
336   // Filter actuator estimation
337   guidance_indi_filter_actuators();
338
339   // Minimum increment in pitch angle, roll angle, thrust and tail rotor input
340   du_min_xinca[0] = -guidance_indi_max_bank - pitch_filt.o[0];
341   du_min_xinca[1] = -guidance_indi_max_bank - roll_filt.o[0];
342   du_min_xinca[2] = (MAX_PPRZ - act_z_filt.o[0]) / c_z_thrust / XINCA_G_SCALING;
343   du_min_xinca[3] = -act_x_filt.o[0] / c_x_tail_rotor / XINCA_G_SCALING;
344   du_min_xinca[3] = -10000;
345
346   // Maximum increment in pitch angle, roll angle, thrust and tail rotor input
347   du_max_xinca[0] = guidance_indi_max_bank - pitch_filt.o[0];
348   du_max_xinca[1] = guidance_indi_max_bank - roll_filt.o[0];
349   du_max_xinca[2] = - act_z_filt.o[0] / c_z_thrust / XINCA_G_SCALING;
350   du_max_xinca[3] = (MAX_PPRZ - act_x_filt.o[0]) / c_x_tail_rotor / XINCA_G_SCALING;
351   du_max_xinca[3] = 10000;
352
353   // Preferred increment in pitch angle, roll angle, thrust and tail rotor input
354   du_pref_xinca[0] = u_pref[0] - pitch_filt.o[0];
355   du_pref_xinca[1] = u_pref[1] - roll_filt.o[0];
356   du_pref_xinca[2] = u_pref[2] - act_z_filt.o[0] / c_z_thrust / XINCA_G_SCALING;
357   du_pref_xinca[3] = u_pref[3] - act_x_filt.o[0] / c_x_tail_rotor / XINCA_G_SCALING;
358
359   // Calculate virtual input
360   v_xinca[0] = a_diff.x;
361   v_xinca[1] = a_diff.y;
362   v_xinca[2] = a_diff.z;
363
364   // WLS Control Allocator
365   uint8_t iter = wls_alloc_guidance(du_xinca, v_xinca, du_min_xinca, du_max_xinca,
366       B, du_prev_xinca, 0, W_acc, W_act, du_pref_xinca, 10000, 10);
367   float_vect_copy(du_prev_xinca, du_xinca, XINCA_NUM_ACT);
368
369   if (iter > max_iter_xinca) {
370     max_iter_xinca = iter;
371   }
372
373   AbiSendMsgTHRUST(THRUST_INCREMENT_ID, du_xinca[2]);
374
375   // Add increment in angles
376   guidance_euler_cmd.theta = pitch_filt.o[0] + du_xinca[0];
377   guidance_euler_cmd.phi = roll_filt.o[0] + du_xinca[1];
378   guidance_euler_cmd.psi = *heading_sp;
379
380   //Add increment in thrust and tail rotor input
381   act_z_in = act_z_filt.o[0] + du_xinca[2] * c_z_thrust * XINCA_G_SCALING;
382   Bound(act_z_in, 0, 9600);
383
384   act_x_in = act_x_filt.o[0] + du_xinca[3] * c_x_tail_rotor * XINCA_G_SCALING;
385   Bound(act_x_in, 0, 9600);
386
387
388 #if GUIDANCE_INDI_RC_DEBUG
389   if (radio_control.values[RADIO_THROTTLE] < 300) {
390     act_z_in = 0;
391   }
```

```
392 #endif
393
394   //Overwrite the thrust command from guidance_v
395   stabilization_cmd[COMMAND_THRUST] = act_z_in;
396
397   //Bound euler angles to prevent flipping
398   Bound(guidance_euler_cmd.phi, -guidance_indi_max_bank, guidance_indi_max_bank);
399   Bound(guidance_euler_cmd.theta, -guidance_indi_max_bank, guidance_indi_max_bank);
400
401   //set the quat setpoint with the calculated roll and pitch
402   struct FloatQuat q_sp;
403   float_quat_of_eulers_yxz(&q_sp, &guidance_euler_cmd);
404   QUAT_BFP_OF_REAL(stab_att_sp_quat, q_sp);
405
406   u_xinca[0] = roll_filt.o[0];
407   u_xinca[1] = pitch_filt.o[0];
408   u_xinca[2] = act_z_filt.o[0];
409   u_xinca[3] = act_x_filt.o[0];
410
411   //Commit tail rotor command
412   if (-stateGetPositionNed_f()->z >= h_thres) {
413     actuators_pprz[4] = act_x_in;
414   } else {
415     actuators_pprz[4] = 0;
416   }
417
418 }
419
420 /**
421  * Filter the thrust, such that it corresponds to the filtered acceleration
422  */
423 void guidance_indi_filter_actuators(void)
424 {
425   // Actuator dynamics
426   act_z = act_z + act_z_dyn * (act_z_in - act_z);
427   act_x = act_x + act_x_dyn * (act_x_in - act_x);
428
429   // Same filter as for the acceleration
430   update_butterworth_2_low_pass(&act_z_filt, act_z);
431   update_butterworth_2_low_pass(&act_x_filt, act_x);
432 }
433
434 /**
435  * Low pass the accelerometer measurements to remove noise from vibrations.
436  * The roll and pitch also need to be filtered to synchronize them with the
437  * acceleration
438  */
439 void guidance_indi_propagate_filters(struct FloatEulers *eulers)
440 {
441   struct NedCoor_f *accel = stateGetAccelNed_f();
442   update_butterworth_2_low_pass(&filt_accel_ned[0], accel->x);
443   update_butterworth_2_low_pass(&filt_accel_ned[1], accel->y);
444   update_butterworth_2_low_pass(&filt_accel_ned[2], accel->z);
445
446   update_butterworth_2_low_pass(&roll_filt, eulers->phi);
447   update_butterworth_2_low_pass(&pitch_filt, eulers->theta);
448 }
449
450 /**
451  * @param Gmat array to write the matrix to [3x3]
452  *
453  * Calculate the matrix of partial derivatives of the pitch, roll and thrust.
454  * w.r.t. the NED accelerations for YXZ eulers
455  * ddx = G*[dtheta,dphi,dT]
456  */
457 void guidance_xinca_calcG_yxz(struct FloatEulers *euler_yxz)
458 {
459
460   // Get rotation matrix from NED to body coordinates
461   struct FloatRMat *ned_to_body_rmat = stateGetNedToBodyRMat_f();
462
```

```
463   // Get vertical body acceleration
464   struct FloatVect3 a_ned = {filt_accel_ned[0].o[0], filt_accel_ned[1].o[0], filt_accel_ned[2].o[0]};
465   struct FloatVect3 a_body;
466   float_rmat_vmult(&a_body, ned_to_body_rmat, &a_ned);
467
468   // Get forward body velocity velocity (ideally airspeed velocity)
469   struct NedCoor_f *ned_speed_f = stateGetSpeedNed_f();
470   struct FloatVect3 speed_ned = {ned_speed_f->x, ned_speed_f->y, ned_speed_f->z};
471   struct FloatVect3 speed_body;
472   float_rmat_vmult(&speed_body, ned_to_body_rmat, &speed_ned);
473
474   // Upward estimate minus gravity is an estimate of the thrust force
475   float T = a_body.z - gravity;
476
477   // Get current attitude angles
478   float sphi = sinf(euler_yxz->phi);
479   float cphi = cosf(euler_yxz->phi);
480   float stheta = sinf(euler_yxz->theta);
481   float ctheta = cosf(euler_yxz->theta);
482
483   // Calculate matrix components
484   G[0][0] = ctheta * cphi * T;
485   G[1][0] = 0;
486   G[2][0] = cphi * (c_l_alpha * 0.5 * rho * speed_body.x * speed_body.x * wing_surface / mass - stheta * T
        );
487   G[0][1] = -stheta * sphi * T;
488   G[1][1] = -cphi * T;
489   G[2][1] = -ctheta * sphi * T;
490   G[0][2] = stheta * cphi;
491   G[1][2] = -sphi;
492   G[2][2] = ctheta * cphi;
493
494   // Only use tail rotor above threshold height
495   if (-stateGetPositionNed_f()->z >= h_thres) {
496     G[0][3] = ctheta;
497     G[1][3] = 0;
498     G[2][3] = -stheta;
499   } else {
500     G[0][3] = 0;
501     G[1][3] = 0;
502     G[2][3] = 0;
503   }
504
505   for (int i = 0; i < XINCA_OUTPUTS; i++) {
506     B[i] = G[i];
507   }
508
509 }
510
511 /**
512  * ABI callback that obtains the acceleration setpoint from telemetry
513  * flag: 0 -> 2D, 1 -> 3D
514  */
515 static void accel_sp_cb(uint8_t sender_id __attribute__((unused)), uint8_t flag, struct FloatVect3 *
        accel_sp)
516 {
517   if (flag == 0) {
518     indi_accel_sp.x = accel_sp->x;
519     indi_accel_sp.y = accel_sp->y;
520     indi_accel_sp_set_2d = true;
521     time_of_accel_sp_2d = get_sys_time_float();
522   } else if (flag == 1) {
523     indi_accel_sp.x = accel_sp->x;
524     indi_accel_sp.y = accel_sp->y;
525     indi_accel_sp.z = accel_sp->z;
526     indi_accel_sp_set_3d = true;
527     time_of_accel_sp_3d = get_sys_time_float();
528   }
529 }
```