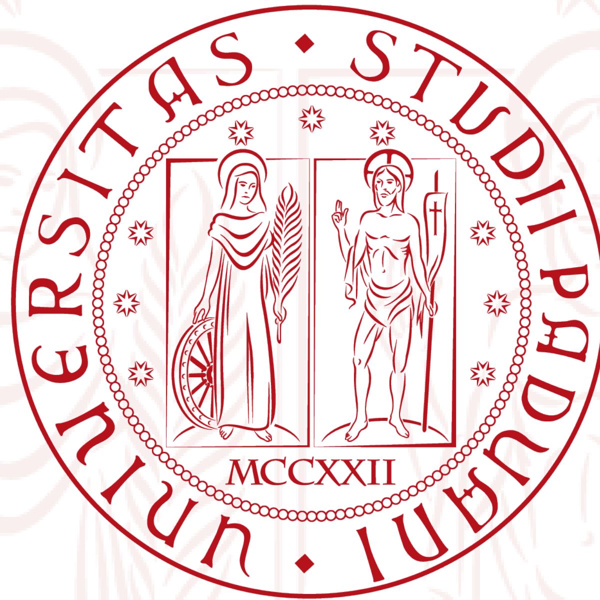
UNIVERSITY OF PADUA



Bachelor Degree in Control Systems

Realization and Co-Simulation of a Three-Loop algorithm for a missile trajectory control

Thursday, September the 27th, 2012

Chair, CS Dept: Pinzoni Stefano
Advisor: Marcuzzi Fabio
Co-Advisor: Da Forno Roberto
Student: Casella Diego

Academic Year 2011/2012

Contents

1 Introduction 5
1.1 About the Thesis . . . 5
1.2 About μLab® software . . . 6

2 Missile 2D model 8
2.1 Linearization . . . 12
2.1.1 $\Delta\dot{\theta}$ calculus . . . 13
2.2 Frequency Domain Considerations . . . 15
2.3 Trim Condition Analysis . . . 16

3 Missile Control strategies 18
3.1 Open-Loop Control . . . 18
3.2 Rate-Gyro Control . . . 19
3.3 Three-Loop Control . . . 20
3.3.1 Phase margin and loop gain . . . 22
3.3.2 Alternate way to analize instability . . . 22

4 Three-Loop matlab simulation 25
4.1 Continuous model . . . 25
4.2 Discretized model . . . 27
4.3 Discretized model - explicit loop delay . . . 29
4.4 Noise disturbance effects . . . 32
4.4.1 Noise disturbance $\sim N(0, 0.25)$ applied to a_c . . . 32
4.4.2 Noise disturbance $\sim N(0, 0.25)$ applied to n_B . . . 33
4.4.3 Noise disturbance $\sim N(0, 0.25)$ applied to $\Delta\dot{\theta}$ . . . 34
4.4.4 Noise disturbance $\sim N(0, 0.25)$ applied to $\Delta\delta$ . . . 35
4.4.5 Noise disturbance $\sim N(0, 0.0025)$ applied to a_c . . . 37
4.4.6 Noise disturbance $\sim N(0, 0.0025)$ applied to n_B . . . 38
4.4.7 Noise disturbance $\sim N(0, 0.0025)$ applied to $\Delta\dot{\theta}$ . . . 39
4.4.8 Noise disturbance $\sim N(0, 0.0025)$ applied to $\Delta\delta$ . . . 40
4.4.9 Noise disturbance applied to $\Delta\delta$, take two . . . 41
4.5 Faulty conditions analysis . . . 43
4.5.1 Absence of n_B . . . 43
4.5.2 Absence of $\Delta\dot{\theta}$ . . . 44
4.6 Moving toward a Co-Simulated approach . . . 45

5 Three-Loop Co-Simulation 46
5.1 Algorithm characterization . . . 46
5.1.1 Input module . . . 47

5.1.2 Limited input values 49
5.1.3 Output module 49
5.1.4 Loop execution time estimation 51
5.2 *μLab* environment setup 53
5.3 Co-Simulation results 55
5.4 Noise disturbance effects 56
5.4.1 Noise disturbance $\sim N(0, 0.25)$ applied to a_c 56
5.4.2 Noise disturbance $\sim N(0, 0.25)$ applied to n_B 57
5.4.3 Noise disturbance $\sim N(0, 0.25)$ applied to $\Delta\delta$ 57
5.4.4 Noise disturbance $\sim N(0, 0.0025)$ applied to a_c 59
5.4.5 Noise disturbance $\sim N(0, 0.0025)$ applied to n_B 60
5.5 Faulty conditions analysis 61
5.5.1 Absence of n_B 61
5.5.2 Absence of $\Delta\dot{\theta}$ 62

6 Conclusions and further development 63

7 Acknowledgements 64

Appendix A Complete Three-Loop Algorithm 67

Appendix B Complete missile_aeroframe_2d script 72

1 Introduction

Provide an accurate and effective model representation of a physical system is a frequently required task for every Control Systems Engineer.
Nonethless, its simulation and authentication process could represent difficult zones to pass through, especially if the system being examinated is composed of both mechanics and electronics components. In fact, in such cases there is no simple answer about how to mathematically treat the electronic devices integrated inside those systems.

If we consider, for example, the well-known problem of the DC motor speed stabilization, there are plenty of books about that topic exposing different approaches to that *mathematical* problem, solved with matlab. Few of them do actually take into consideration *how* the controller is being implemented on an electronic device but, simulating the firmware *and* the physical system under the same application, is an approach nobody undertake ever.
Co-Simulation allows us to overcome the aforementioned problems, achieving higher levels of model accuracy by simulating the interactions between the embedded firmware of the controller and the physical model being controlled, bringing to light aspects that were understimated or even not esteemed at all. For this purpose, we will use the *μLab* co-simulation software[1], which grants us the ability to evaluate how the whole system evolves.

1.1 About the Thesis

The purpose of this thesis is to implement a piece of software that realizes the so-called *Three-Loop Algorithm*, which aims at stabilizing the trajectory of a missile over a given flight path, and verify its correctness with the missile model provided by means of a Co-Simulation approach. Therefore, in Chapther 2 we will first estimate the equations ruling the law of motion for the missile system then, in Chapter 3, we will analize the mathematical expression of the *Three-Loop Algorithm.* In Chapter 4, the results of the complete matlab model simulation will be exposed,in both the absence and presence of various sources of disturbances, and we are also going to analize how the controller behaves when faulty conditions occur. We will also discuss the limitations and obstacles that are not easy to spot or to model with a pure matlab approach, which will consequentely lead us to the co-simulated world. Then, in Chapter 5, we will implement the C code for the *Three-Loop Algorithm*, designed specifically for being executed by the Microchip's

[1] For a comprehensive explanantion on how it works, see subsection 1.2

PIC18F4620 microcontroller. In that chapter we will see how the *real* code interacts with the matlab missile model, thanks to μ*Lab* co-simulation software. Starting from a naive implementation, which is a simple transposition in C code of the *Three-Loop Algorithm*, we will pass through an iterative process which will show us the obstacles involved at each step, the solutions adopted to overcome them, and the results obtained each time.

1.2 About μLab® software

As we already mentioned, μLab provides an environment specifically designed to ease Engineers' study about the interactions between the firmware, and the mathematical model represeting the system for which the firmware is written. This is how μLab interface looks like:

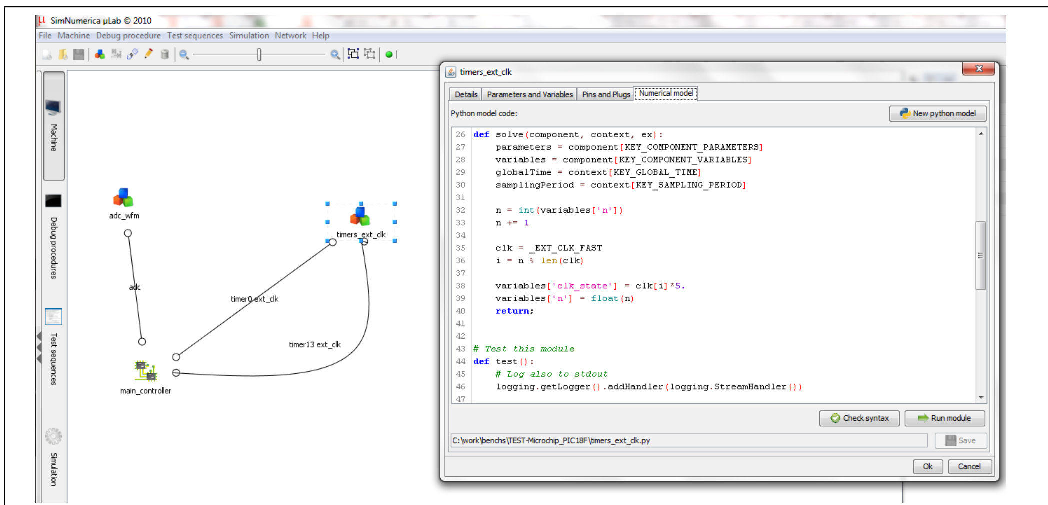


Figure 1: μLab interface.

As you can see, it pretty much resembles Simulink, although the way μLab works behind the scenea is different compared to how Simulink does it. The blue/red/green blocks in μLab *Machine* view, known as *component models*, contain a standard Matlab or Python function[2], and may define a set of parameters and physical variables needed to be passed to the μLab environment. Every component model sends/receives its parameter and variables to/from other component models by means of *links*: unlike Simulink, μLab's links can connect parameters or variables of the same nature only(i.e. voltage, current, temperature, frequency etc...). This is because in muLab the simulation is enforced to have a physical meaning, not abstract.

[2]which is solved by their respective interpreters.

The grey block, called *embedded platform*, is the key component of μLab. It's purpose is to behave as a real microcontroller would do, so it provides the same set of input/output peripherals, stores the same binary firmware used to flash the real electronic device, and executes it as the real controller does. That's where the real power of μLab comes to light: when the simulation starts it executes, in a 1:1 fashion, every assembler instruction contained in the binary file, triggers the affected pins, and forwards their state to the other component models connected so the other components can use those values to complete the whole simulation. In this way, it is possible to perform a true *co-simulation* between the firmware that will be hosted in the controller and the mathematical model, where the firmware gets executed as it would happen in the real device, and its effects are forwarded to the Matlab (or Python) models and solved with its specific tools.
It also has other interesting features, such as the ability to track and save the value of each memory location of the microcontroller, trigger pre-defined actions when a given register reach a specified value and, the most interesting of all of them, the Python routine substitution, which speeds up firmware prototyping by replacing a given firmware routine [3] with a Python script, without the need to recompile the sourcecode at all.

[3] Or even the entire firmware

2 Missile 2D model

In the following paragraph we are going to briefly illustrate the physics involved with the missile model, in two dimension, courtesy of Da Forno's slides[6] [7]. These assumptions could be expanded to cover the general 3D case.

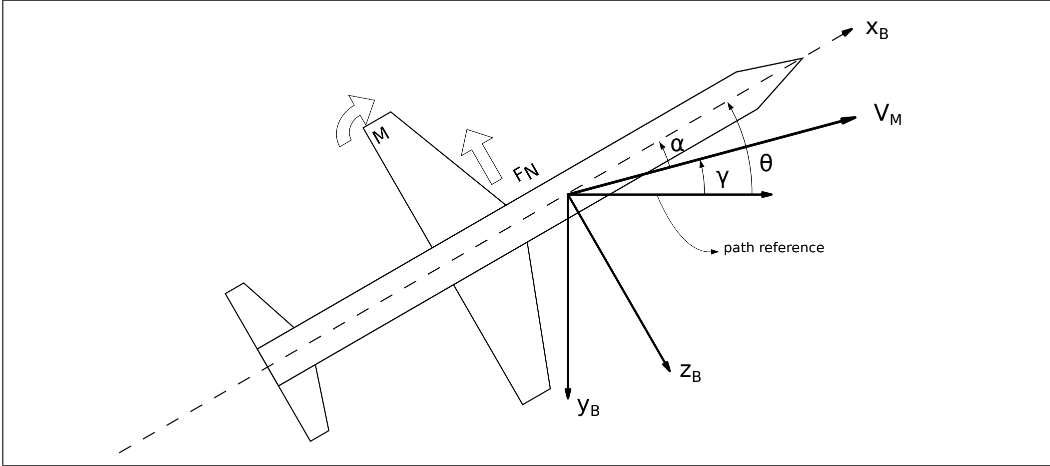


Figure 2: Missile representation.

The variables appearing in Figure 2 have the following meaning:

- F_N, the normal force applied to the missile, which keeps the missile flying, measured in $[N]$;
- M, the rotational inertia acting on the missile, responsible for the trajectory changes, measured in $[kg \cdot m^2]$.
- (x_B, y_B, z_B) represent the local coordinate system, centered on the missile's center of mass;
- V_M, the missile's speed, expressed in $[Mach]$;
- α, the angle of attack $[rad]$;
- γ,the flight path angle $[rad]$;
- θ, the pitch of the missile $[rad]$;

First, we need to determine the equations governing the run of F_N and M. The general expression of F_N is given by the following equation

$$F_N = Q \cdot S_{Ref} \cdot C_N \tag{2.1}$$

in which Q equals to

$$Q = \frac{1}{1} \cdot \rho \cdot V_M^2 \tag{2.2}$$

and the variables appearing are, consequentely:

- ρ, the istantaneous radius of the circumference tangent to the trajectory, see Figure 5;
- S_{Ref}, the reference area;
- C_N, the normal force coefficient.

Before proceding any further, in Figure 3 and 4 we will also illustrate the missile's plant, and the detailed view of the wing and tail fin.

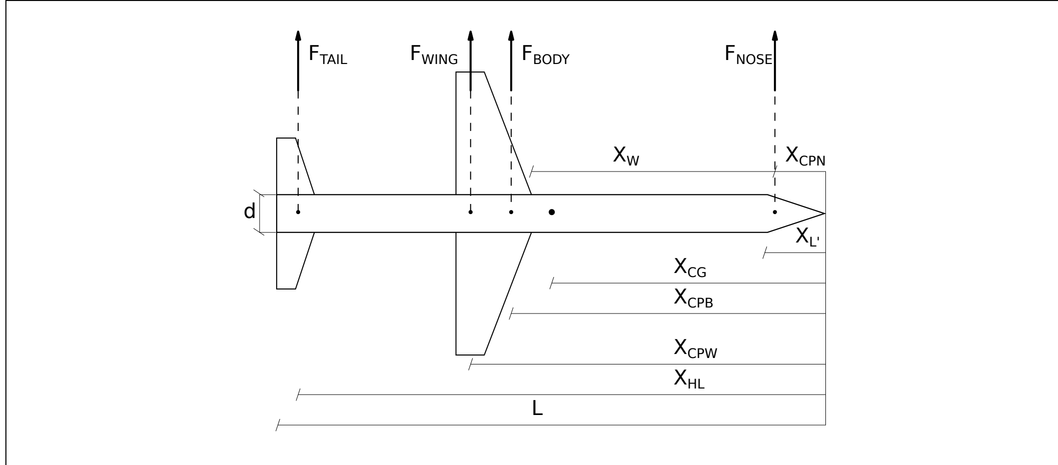


Figure 3: Missile's plant.

Now we have everything needed to calculate C_N:

$$C_N = \underbrace{2\alpha}_{nose} + \underbrace{\frac{1,5 \cdot S_{Plan}\alpha^2}{S_{Ref}}}_{body} + \underbrace{\frac{8 \cdot S_W \cdot \alpha}{\beta \cdot S_{Ref}}}_{wings} + \underbrace{\frac{8 \cdot S_T(\alpha+\delta)}{\beta \cdot S_{Ref}}}_{tailfin} \tag{2.3}$$

The general equation for the rotational inertia M is :

$$M = Q \cdot S_{Ref} \cdot C_M \cdot d \tag{2.4}$$

We are going to decompose the torque for each missile section, that is:

$$M_{Nose} = F_{M_{Nose}} \cdot (x_{CG} - x_{CPN}) = 2\alpha \cdot Q \cdot S_{Ref} \cdot \overbrace{(x_{CG} - x_{CPN})}^{>0} \tag{2.5}$$

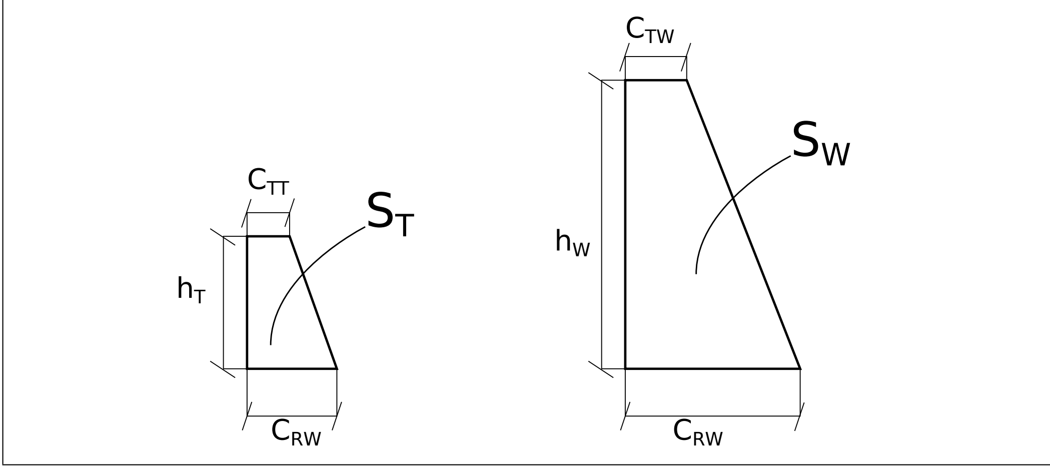


Figure 4: Detailed view of the wing and tail fin.

$$M_{Body} = F_{M_{Body}} \cdot (x_{CG} - x_{CPB}) = \frac{1,5 \cdot S_{Plan} \alpha^2}{S_{Ref}} \cdot Q \cdot S_{Ref} \cdot \overbrace{(x_{CG} - x_{CPB})}^{<0} \tag{2.6}$$

$$M_{Wing} = \frac{8 \cdot S_W \cdot \alpha}{\beta \cdot S_{Ref}} \cdot Q \cdot S_{Ref} \cdot \overbrace{(x_{CG} - x_{CPU})}^{<0} \tag{2.7}$$

$$M_{Tail} = \frac{8 \cdot S_T (\alpha + \delta)}{\beta \cdot S_{Ref}} \cdot Q \cdot S_{Ref} \cdot \overbrace{(x_{CG} - x_{HL})}^{<0} \tag{2.8}$$

and now we can sum everything up, and finally obtain C_M:

$$C_M = \frac{M_{Nose} + M_{Body} + M_{Wing} + M_{Tail}}{Q \cdot S_{Ref} \cdot d} \tag{2.9}$$

The standard values and proportions are:

$$\begin{cases} x_{CPM} = 0,67 \cdot L' \\ x_{CPW} = L' + x_W + 0,7 \cdot C_{RW} - 0,2 \cdot C_{TW} \\ A_N = 0,67 \cdot L' \cdot d \\ A_B = (L - L') \cdot d \\ x_{CPB} = \dfrac{0,67 \cdot A_N \cdot L' + A_B \cdot (L' - 0,5(L - L'))}{A_N + A_B} \\ x_{CG} \simeq 0,5 \cdot L \text{ (simplified)} \end{cases} \tag{2.10}$$

Since the missile trajectory control is performed by adjusting its acceleration, we need to determine it first:

$$n_B = \frac{F_N}{m} = \frac{C_N \cdot Q \cdot S_{Ref}}{m} \tag{2.11}$$

or, if we know the weight of the missile instead of its mass

$$n_B = \frac{C_N \cdot Q \cdot S_{Ref} \cdot g}{w} \tag{2.12}$$

The rotational inertia equation[6] becomes

$$\ddot{\theta} = \frac{M}{I_y} = \frac{Q \cdot S_{Ref} \cdot d \cdot C_M}{I_y} \tag{2.13}$$

Considerations about α and γ Consider the following figure:

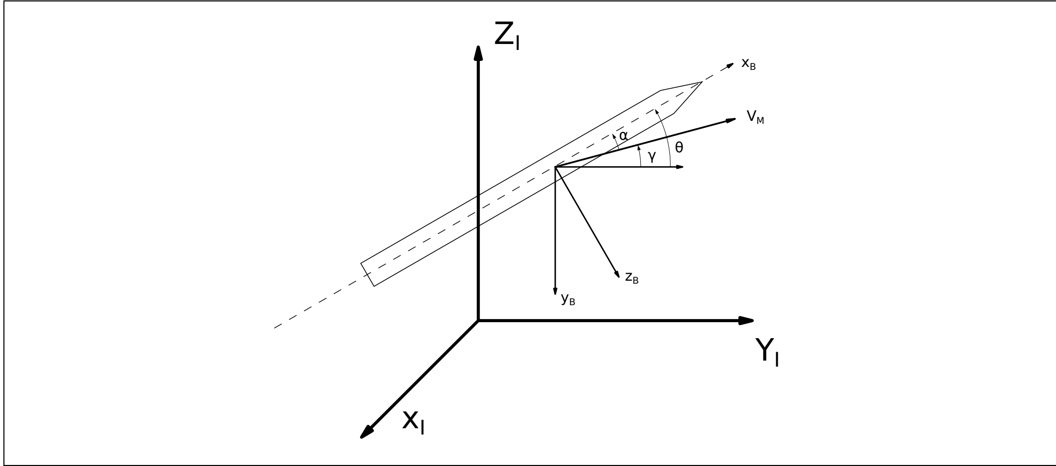


Figure 5: Schematic of the missile

The question we are interested in is: what is responsible of the missile's trajectory change? - *The force normal to the trajectory* -
V_M is always tangent to the CG trajectory, so every trajectory change is performed by supplying an acceleration normal to V_M. According to Figure 6 :

- ρ is radius of the circle istantaneously tangent to the trajectory;
- $\dot{\gamma}$ denotes istantaneous angular speed (also, the angular speed the V_M is spinnig with);
- a_c is centripetal acceleration, responsible of the trajectory change.

with a_c given from the equation below:

$$a_c = \dot{\gamma}^2 \cdot \rho = \dot{\gamma} \cdot \underbrace{\dot{\gamma} \cdot \rho}_{V_M} \tag{2.14}$$

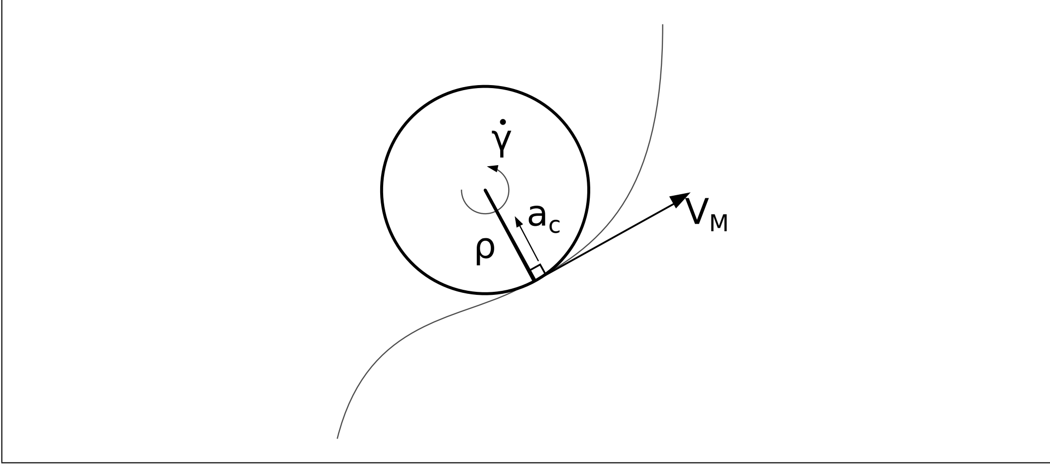


Figure 6: Scheme of the istantaneous acceleration.

If we introduce the notation $n_L = a_c$, then (2.14) can be rewritten as

$$n_L = \dot{\gamma} \cdot V_M \Longrightarrow \boxed{\dot{\gamma} = \frac{n_L}{V_M}} \tag{2.15}$$

If the angle of attack α is sufficient small ($< 15°$), n_L could be easily substituted by n_B, the normal acceleration of the veichle's axis. Hence

$$\alpha = \theta - \gamma \Longrightarrow \dot{\alpha} = \dot{\theta} - \dot{\gamma} = \begin{cases} \dot{\theta} - \dfrac{n_L}{V_M} & \text{generic form} \\ \\ \dot{\theta} - \dfrac{n_B}{V_M} & \text{when } \alpha < 15° \end{cases} \tag{2.16}$$

2.1 Linearization

Now, we need to describe the following relationships

$$\xrightarrow{n_L} \boxed{(?)} \xrightarrow{\dot{\gamma}} \text{ and } \overset{\alpha}{\underset{\delta}{\rightrightarrows}} \boxed{(?)} \xrightarrow{\dot{\gamma}}$$

by recalling the equation (2.15)

$$\dot{\gamma} = \frac{n_L}{V_M} \simeq \frac{n_B}{V_M} \tag{2.17}$$

By recalling equation (2.12), and estimating Δn_B, we get

$$\Delta n_B = \frac{Q \cdot S_{Ref}}{m} \cdot \underbrace{\left(\frac{\partial C_M}{\partial \alpha} \cdot \Delta\alpha + \frac{\partial C_M}{\partial \delta} \cdot \Delta\delta \right)}_{\triangleq \Delta C_M} \tag{2.18}$$

$\left(\frac{\partial C_M}{\partial \alpha}, \frac{\partial C_M}{\partial \delta}\right)$ are evaluated in the missile equilibrium, called *Trim Condition*, taking place when $(F_N, M) = (0, 0)$. Thus

$$\begin{cases} \dfrac{\partial C_M}{\partial \alpha} = 2 + \dfrac{1,5 \cdot S_{Plan} \cdot 2 \cdot \alpha}{S_{Ref}} + \dfrac{8 \cdot S_W}{\beta \cdot S_{Ref}} + \dfrac{8 \cdot S_T}{\beta \cdot S_{Ref}} & (2.19a) \\ \dfrac{\partial C_M}{\partial \delta} = \dfrac{8 \cdot S_T}{\beta \cdot S_{Ref}} & (2.19b) \end{cases}$$

We also define

$$\left(\frac{\partial C_M}{\partial \alpha}, \frac{\partial C_M}{\partial \delta}\right) \triangleq (C_{N_\alpha}, C_{N_\delta}) \quad (2.20)$$

Combining (2.18) and the previous definition with (2.17) we obtain

$$\begin{aligned} \Delta\dot{\gamma} &= \frac{\Delta u_B}{V_M} = \frac{Q \cdot S_{Ref}}{m} \cdot (C_{N_\alpha} \Delta\alpha + C_{N_\delta} \Delta\delta) \\ &= -z_\alpha \cdot \Delta\alpha - z_\delta \cdot \Delta\delta \end{aligned} \quad (2.21)$$

From Figure 2, it's easy to see that

$$\Delta\alpha = \Delta\theta - \Delta\gamma \Longrightarrow \Delta\dot{\alpha} = \Delta\dot{\theta} - \Delta\dot{\gamma} \quad (2.22)$$

hence, by applying equation (2.21), it yelds the result below

$$\boxed{\Delta\dot{\alpha} = \Delta\dot{\theta} + z_\alpha \cdot \Delta\alpha + z_\delta \cdot \Delta\delta} \quad (2.23)$$

2.1.1 $\Delta\dot{\theta}$ calculus

By applying to (2.13) the same process that lead us to (2.21), we obtain

$$\Delta\dot{\theta} = \frac{\Delta M}{I_y} = \frac{Q \cdot S_{Ref} \cdot d}{I_y} \cdot \Delta C_M \quad (2.24)$$

where ΔC_M has already been defined in (2.18). Therefore

$$\begin{aligned} \Delta\ddot{\theta} &= \frac{Q \cdot S_{Ref} \cdot d}{I_y} \cdot \left(\frac{\partial C_M}{\partial \alpha} \cdot \Delta\alpha + \frac{\partial C_M}{\partial \delta} \cdot \Delta\delta\right) \\ &= M_\alpha \cdot \Delta\alpha + M_{delta} \cdot \Delta\delta \end{aligned} \quad (2.25)$$

with obvious meaning of the terms M_α and M_δ. If we apply the Laplace transform on the above formula, the result is

$$s \cdot \Delta\dot{\theta}(s) = M_\alpha \cdot \Delta\alpha(s) + M_\delta \cdot \Delta\delta(s) \quad (2.26)$$

substituting the previous result inside equation (2.23) gives us

$$\Delta\dot{\alpha} = \frac{1}{s}\left(M_\alpha \cdot \Delta\alpha(s) + M_\delta \cdot \Delta\delta(s)\right) \tag{2.27}$$

and, if we multiply by the continuos integrator $\frac{1}{s}$, we get

$$\Delta\alpha = \frac{1}{s^2}\left(M_\alpha \cdot \Delta\alpha(s) + M_\delta \cdot \Delta\delta(s)\right) + \frac{1}{s} \cdot \left(z_\alpha \cdot \Delta\alpha + z_\delta \cdot \Delta\delta\right) \tag{2.28}$$

After some algebraic steps, left as exercise for the reader, the final formula yelds

$$\boxed{\Delta\alpha = \frac{s \cdot z_\delta + M_\delta}{s^2 - s \cdot z_\alpha - M_\alpha} \cdot \Delta\delta} \tag{2.29}$$

Furthermore, by substituting (2.29) in (2.21)

$$\Delta\dot{\gamma} = -\left(z_\alpha \frac{s \cdot z_\delta + M_\delta}{s^2 - s \cdot z_\alpha - M_\alpha} + z_\delta\right) \cdot \Delta\delta \tag{2.30}$$

Now we are finally able to tie the missile acceleration n_B and the tail fin angle variation, that is

$$\Delta n_B = V_M \cdot \Delta\dot{\gamma} = -V_M\left(z_\alpha \frac{s \cdot z_\delta + M_\delta}{s^2 - s \cdot z_\alpha - M_\alpha} + z_\delta\right) \cdot \Delta\delta \tag{2.31}$$

At this point it is fairly easy to retrieve the transfer function between Δn_B and $\Delta\delta$. After some algebraic calculus and some handy definitions, we get the following transfer function:

$$\boxed{\frac{\Delta n_B}{\Delta\delta} = k_1 \cdot \frac{1 - \dfrac{s^2}{\omega_Z^2}}{\dfrac{s^2}{\omega_{AF}^2} + \dfrac{2 \cdot \xi_{AF}}{\omega_{AF}} \cdot s + 1}} \tag{2.32}$$

where

$$\begin{cases} k_1 = -V_M \cdot \dfrac{z_\delta \cdot M_\alpha - z_\alpha \cdot M_\delta}{M_\alpha} \\ \\ \omega_Z^2 = \dfrac{z_\delta \cdot M_\alpha - z_\alpha \cdot M_\delta}{z_\delta} \\ \\ \omega_{AF} = \sqrt{-M_\alpha} \\ \\ \xi_{AF} = \dfrac{z_\alpha \cdot \omega_{AF}}{2 \cdot M_\alpha} \end{cases} \tag{2.33}$$

2.2 Frequency Domain Considerations

We are interested in analizing the frequency response of (2.32), therefore we performed the following substitution

$$s \longrightarrow i\omega, \qquad i = \sqrt{-1} \tag{2.34}$$

obtaining

$$\frac{\Delta n_B}{\Delta\delta}(i\omega) = k_1 \cdot \frac{1 + \dfrac{\omega^2}{\omega_Z^2}}{-\dfrac{\omega^2}{\omega_{AF}^2} + i\omega \cdot \dfrac{2 \cdot \xi_{AF}}{\omega_{AF}} + 1} \tag{2.35}$$

If $\omega = 0$

$$\frac{\Delta n_B}{\Delta\delta} = k_1 \Longrightarrow \Delta\delta = \frac{1}{k_1}\Delta nB \tag{2.36}$$

In general,

$$\begin{aligned} \left|\frac{\Delta n_B}{\Delta\delta}(i\omega)\right| &= k_1 \cdot \frac{\left|1 + \frac{\omega^2}{\omega_Z^2}\right|}{\left|\left(1 - \frac{\omega^2}{\omega_{AF}^2}\right) + i \cdot \frac{\omega \cdot 2 \cdot \xi_{AF}}{\omega_{AF}}\right|} \\ &= k_1 \cdot \frac{\left(1 + \frac{\omega^2}{\omega_Z^2}\right)}{\sqrt{\left(1 - \frac{\omega^2}{\omega_{AF}^2}\right)^2 + \left(\frac{\omega \cdot 2 \cdot \xi_{AF}}{\omega_{AF}}\right)^2}} \end{aligned} \tag{2.37}$$

With the following Bode plot of the phase

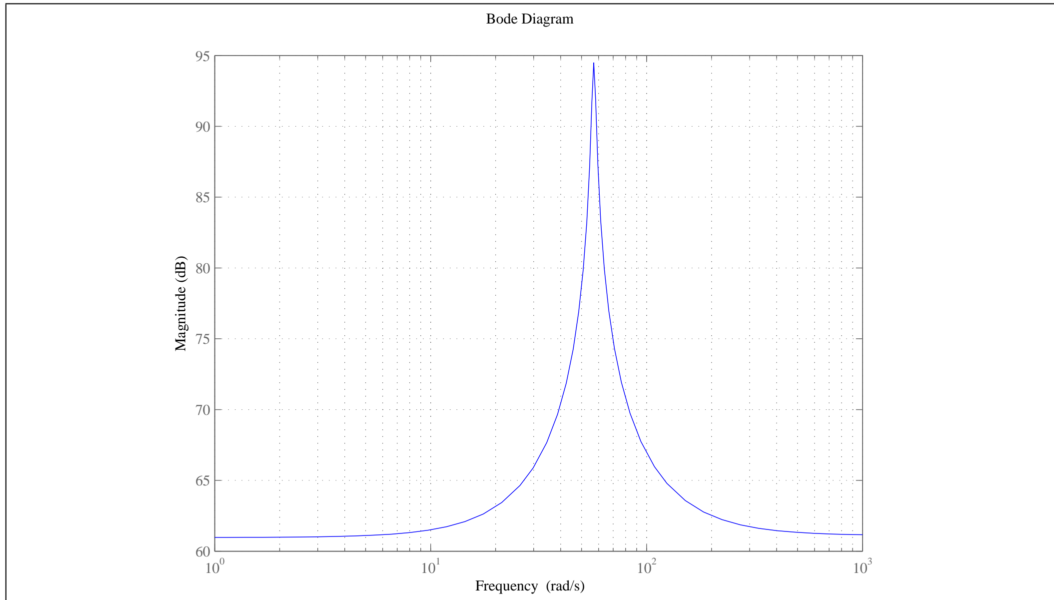


Figure 7: Bode diagram of $\left|\frac{\Delta n_B}{\Delta\delta}(i\omega)\right|$.

2.3 Trim Condition Analysis

We need to find out (α, δ) in order to compensate F_N and balance M. Recalling the formula (2.12)

$$F_N = m \cdot a = m \cdot u_B = Q \cdot S_{Ref} \cdot C_{M_{Trim}} \tag{2.38}$$

hence

$$C_{M_{Trim}} = \frac{m \cdot u_B}{Q \cdot S_{Ref}} \tag{2.39}$$

where, under *Trim* condition, $u_B = g \cdot cos(\theta)$. It must also be valid the equation

$$C_{M_{Trim}} = y_1 \cdot \alpha_T + y_2 \cdot \alpha_T^2 + y_3 \cdot \delta_T \tag{2.40}$$

$$\begin{cases} y_1 = 2 + \dfrac{8 \cdot S_W}{\beta \cdot S_{Ref}} + \dfrac{8 \cdot S_{WT}}{\beta \cdot S_{Ref}} \\ \\ y_2 = \dfrac{1,5 \cdot S_{Plan}}{S_{Ref}} \\ \\ y_3 = \dfrac{8 \cdot S_T}{\beta \cdot S_{Ref}} \end{cases} \tag{2.41}$$

In order to balance torques, it must be $C_M = 0$, hence

$$C_M = 0 = y_4 \cdot \alpha_T + y_5 \cdot \alpha_T^2 + y_6 \cdot \delta_T \tag{2.42}$$

$$\begin{cases} y_4 = \dfrac{2\,(x_{CG} - x_{CPM})}{d} + \dfrac{8 \cdot S_W\,(x_{CG} - x_{CPW})}{\beta \cdot S_{Ref} \cdot d} + \dfrac{8 \cdot S_T\,(x_{CG} - x_{HL})}{\beta \cdot S_{Ref} \cdot d} \\ \\ y_5 = \dfrac{1,5 \cdot S_{Plan}\,(x_{CG} - x_{CPB})}{S_{Ref} \cdot d} \\ \\ y_6 = \dfrac{8 \cdot S_T\,(x_{CG} - x_{HL})}{S_{Ref} \cdot d} \end{cases} \tag{2.43}$$

We have now two equations, with six variables:

$$\begin{cases} y_1 \cdot \alpha_T + y_2 \cdot \alpha_T^2 + y_3 \cdot \delta_T = C_{M_{Trim}} \\ \\ y_4 \cdot \alpha_T + y_5 \cdot \alpha_T^2 + y_6 \cdot \delta_T = 0 \end{cases} \tag{2.44}$$

if we solve δ_T from the second equation, and put its result in the first one, we get:

$$\delta_T = \frac{-y_4\alpha_T - y_5\alpha_T^2}{y_6} \tag{2.45}$$

$$C_{M_{Trim}} = y_1 \cdot \alpha_T + y_2 \cdot \alpha_T^2 + y_3 \cdot \left(\frac{-y_4\alpha_T - y_5\alpha_T^2}{y_6}\right) \tag{2.46}$$

after some algebric manipulation, we get

$$\alpha_T^2 \underbrace{\left(y_2 - \frac{y_3 \cdot y_5}{y_6}\right)}_{p_2} + \alpha_T \underbrace{\left(y_1 - \frac{y_3 \cdot y_4}{y_6}\right)}_{p_3} - C_{N_{Trim}} = 0 \tag{2.47}$$

and the system of two equations (2.44) can be rewritten[4] as

$$\begin{cases} \alpha_T = \dfrac{-p_3 + \sqrt{p_3^2 + 4p_2 C_{N_{Trim}}}}{2p_2} \\ \\ \delta_T = \dfrac{-y_4\alpha_T - y_5\alpha_T^2}{y_6} \end{cases} \tag{2.48}$$

In an analogous way, we also obtain

$$\boxed{\frac{\Delta\dot{\theta}}{\Delta\delta} = k_3 \frac{1 + T_\alpha \cdot s}{\dfrac{s^2}{\omega_{AF}^2} + \dfrac{2\xi_{AF}}{\omega_{AF}} \cdot s + 1}} \tag{2.49}$$

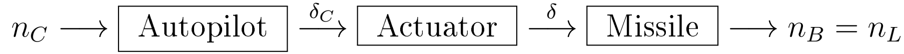
where

$$\begin{cases} k_3 = \dfrac{k_1}{V_M} \\ \\ T_\alpha = \dfrac{M_\delta}{M_\alpha z_\delta - M_\delta z_\alpha} \end{cases} \tag{2.50}$$

[4] as you may have noticed, instead of the standard $\pm$ solution, we kept only the additive one since it's the only one that has a physical sence.

3 Missile Control strategies

Conceptually, the blocks scheme we are going to implement looks like the figure below:



where

- n_C is the commanded acceleration, which is derived from the tracking strategy adopted according with the target type:
  - fixed target
  - moving target
- δ_C is the commanded fin angle generated by the autopilot block;
- δ is the effective fin angle;
- n_L is the normal acceleration needed to perform the tracking.

The most common control stategies are, in increasing order of accuracy and complexity:

- Open-Loop Control;
- Rate-Gyro Control;
- Three-Loop Control.

3.1 Open-Loop Control

This is the easiest of the three type of controller we are going to discuss: by assuming low operational frequencies[5], the equation (2.32) can be simplified obtaining

$$\frac{\Delta nB}{\Delta\delta} \simeq k_1 \Longrightarrow \Delta\delta \simeq \frac{1}{k_1}\cdot \Delta n_B \tag{3.1}$$

We are not discussing any further this rudimentary controller because of the poor performances offered.

[5]that is, $\omega < \omega_{AF}$.

3.2 Rate-Gyro Control

To improve the poor performances of the Open-Loop controller[6], we introduced a feedback from the angular velocity $\Delta\dot{\theta}$ retrieved with a rate-gyro.

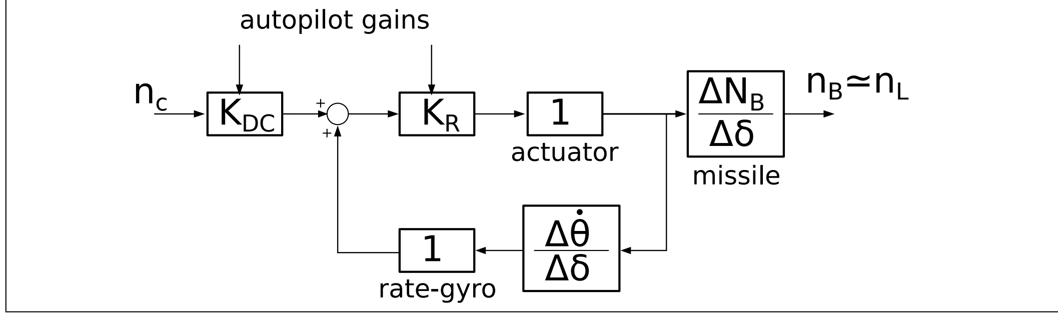


Figure 8: Rate-Gyro schematic

As we have already seen in (2.32) and (2.49),

$$\frac{\Delta n_B}{\Delta\delta} = k_1 \cdot \frac{1 - \dfrac{s^2}{\omega_Z^2}}{\dfrac{s^2}{\omega_{AF}^2} + \dfrac{2 \cdot \xi_{AF}}{\omega_{AF}} \cdot s + 1}$$

$$\frac{\Delta\dot{\theta}}{\Delta\delta} = k_3 \cdot \frac{1 + T_\alpha \cdot s}{\dfrac{s^2}{\omega_{AF}^2} + \dfrac{2 \cdot \xi_{AF}}{\omega_{AF}} \cdot s + 1}$$

If we substitute those transfer functions in the block above, the global transfer function we get is

$$\boxed{\frac{n_B}{n_C} = \frac{K_{DC} \cdot k_1 \cdot K_R}{1 - K_R \cdot k_3} \cdot \frac{1 - \dfrac{s^2}{\omega_Z^2}}{\dfrac{s^2}{\omega_{AF}^2 \cdot (1 - K_R \cdot k_3)} + \dfrac{\dfrac{2 \cdot \xi_{AF}}{\omega_{AF}} - K_R \cdot k_3 \cdot T_\alpha}{1 - K_R \cdot k_3} \cdot s + 1}} \tag{3.2}$$

If want to achieve in steady state $n_B = n_C$, it must be verified the following

$$\frac{K_{DC} \cdot k_1 \cdot K_R}{1 - K_R \cdot k_3} \Longrightarrow \boxed{K_{DC} = \frac{1 - K_R \cdot k_3}{k_1 \cdot K_R}} \tag{3.3}$$

[6] which also has a very low damping factor.

The natural frequency ω_n and its damping factor ξ are:

$$\begin{cases} \omega_n = \omega_{AF}\sqrt{1 - K_R\cdot k_3} \\ \\ \xi = \dfrac{\omega_n}{2}\cdot \dfrac{\dfrac{2\cdot \xi_{AF}}{\omega_{AF}} - K_R\cdot k_3\cdot T_\alpha}{1 - K_R\cdot k_3} \end{cases} \tag{3.4}$$

Usually, $K_R \simeq 0,1$.

3.3 Three-Loop Control

The complete scheme is the following:

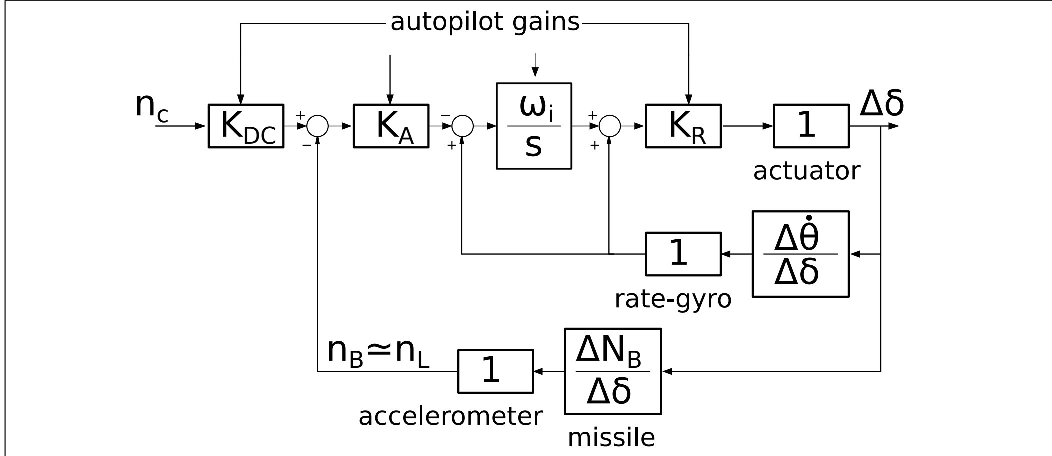


Figure 9: *Three-Loop Algorithm* schematic

The Simulink implementation is easy but, on the other hand, the Matlab one is more complicated and requires a time domain representation

$$\left\{\left[(n_C\cdot K_{DC} - \Delta n_B)(-K_A) + \Delta\dot{\theta}\right]\frac{\omega_i}{s} + \Delta\dot{\theta}\right\} K_R = \Delta\delta \tag{3.5}$$

that can be rewritten as

$$\left[\frac{-(n_C\cdot K_{DC} - \Delta n_B)K_A + \Delta\dot{\theta}}{s}\omega_i + \Delta\dot{\theta}\right] K_R = \Delta\delta \tag{3.6}$$

Now it is possible to inverse *Laplace*-transform the above equation, and obtain

$$\left\{\omega_i \int_0^t \left[\Delta\dot{\delta}(\tau) - (n_C\cdot K_{DC} - \Delta n_B)K_A d\tau + \Delta\dot{\theta}(t)\right]\right\} K_R = \Delta\delta(t) \tag{3.7}$$

if we define

$$a(t) \triangleq \int_0^t \left[\Delta\dot{\delta} - (n_C \cdot K_{DC} - \Delta n_B)K_A\right] dt \tag{3.8}$$

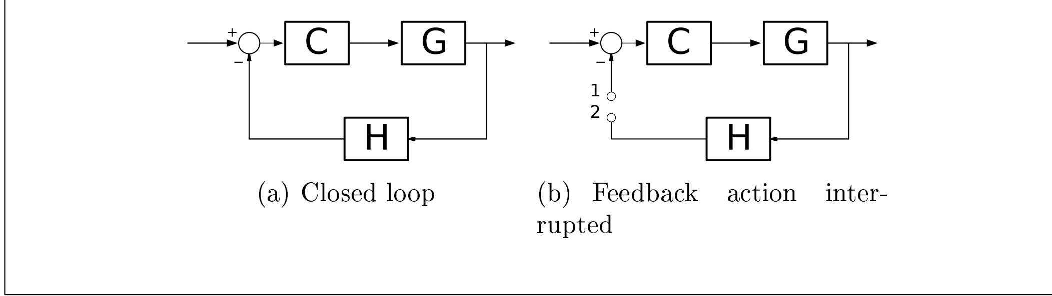
we would notice that its derivative

$$\dot{a}(t) = \left[\Delta\dot{\delta} - (n_C \cdot K_{DC} - \Delta n_B)K_A\right] \tag{3.9}$$

is the differential equation that is gonna be integrated along with the missile dynamics.

3.3.1 Phase margin and loop gain

Assuming the standard feedback scheme recalled in Figure 10, if the feedback



(a) Closed loop (b) Feedback action interrupted

Figure 10: Feedback scheme references

action gets interrupted as shown in Figure 10(b) , supposing in (1) does exists the signal g_1 and removing the reference signal, in (2) we will have

$$g_2(s) = (-g_1(s))C(s)G(s)H(s) \tag{3.10}$$

we can conclude that if

$$\begin{cases} |C(s)G(s)H(s)| = 1 \\ \angle(C(s)G(s)H(s)) = -180° \end{cases} \implies g_1 \equiv g_2 \tag{3.11}$$

which means the signal is self-substaining and there is presence of an oscillating component. It is necessary to have a sufficient phase margin and loop gain that satisfy the following

$$\begin{cases} |C(i\omega)G(i\omega)H(i\omega)| = 1 \\ \angle(C(i\omega)G(i\omega)H(i\omega)) \leqslant -180° + \phi \end{cases} \tag{3.12}$$

or

$$\begin{cases} |C(i\omega)G(i\omega)H(i\omega)| = \dfrac{1}{n} \\ \angle(C(i\omega)G(i\omega)H(i\omega)) = -180° \end{cases} \tag{3.13}$$

Standard values for stabilization are $n \simeq 2$ and $\phi \simeq 45°$.

3.3.2 Alternate way to analize instability

According to Figure 11, the transfer function between $X(s)$ and $d(s)$ is

$$X(s) = \frac{1}{1 + H(s)G(s)C(s)} d(s) \tag{3.14}$$

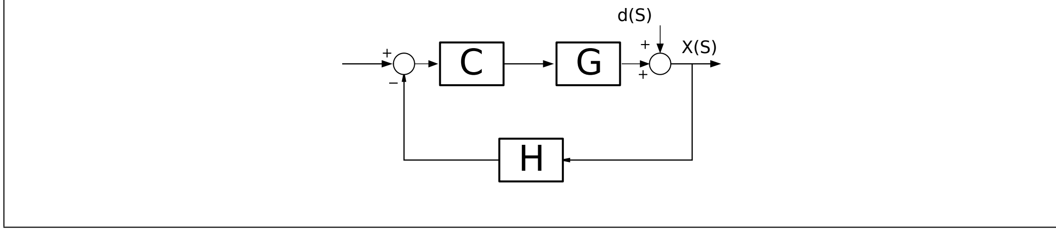


Figure 11: Closed-loop controller with noise

in frequency domain

$$X(i\omega) = \frac{1}{1 + H(i\omega)\ G(i\omega)C(i\omega)} d(i\omega) \tag{3.15}$$

if $HCG(i\omega) = -1 \Longrightarrow |X(i\omega)| \longrightarrow \infty$, which leads us to the previous result:

$$HCG(i\omega) = -1 \Longrightarrow \begin{cases} |HCG| = 1 \\ \angle(HCG) = -180° \end{cases} \tag{3.16}$$

By imposing stability between pins (1) and (2) and the following project constraints

- $\omega_{CR} = 50\ [rad/s]$;
- $\xi = 0,3$;
- $\tau = 0,1\ [s]$

now we are finally able to determine all the parameters of the block scheme.

$$\omega = \frac{\tau\omega_{CR}\left(1 + \dfrac{2\xi_{AF}\omega_{AF}}{\omega_{CR}}\right) + 1}{2\xi\tau} \tag{3.17}$$

$$\omega_0 = \frac{\omega}{\sqrt{\tau\omega_{CR}}} \tag{3.18}$$

$$\xi_0 = \frac{1}{2}\omega_0 \left[\frac{2\xi}{\omega} + \tau - \frac{\omega_{AF}^2}{\omega_{CR}\omega_0^2}\right] \tag{3.19}$$

which lead us to

$$\boxed{K_A = \frac{k_3}{k_C k_1}} \tag{3.20}$$

$$\boxed{\omega_i = \frac{T_\alpha K_C \omega_0^2}{1 + K_C + \dfrac{\omega_0^2}{\omega_Z^2}}} \tag{3.21}$$

Furthermore,

$$K_0 = -\frac{\omega^2}{\tau\omega_{AF}^2} \tag{3.22}$$

$$K = \frac{K_0}{k_1(1+K_C)} \tag{3.23}$$

which allows us to calculate the last two blocks gains

$$\boxed{K_R = \frac{K}{K_A\omega_i}} \tag{3.24}$$

$$\boxed{K_{DC} = 1 + \frac{1}{K_A V_M}} \tag{3.25}$$

4 Three-Loop matlab simulation

After modeling the entire system, it is now time to confirm the overall quality of the assumptions previously made. Therefore, we will first simulate the continuous model and then we will proceed to morph the *Three-Loop* section into its discrete, real counterpart. At the end of this chapter, we will finally recap and briefly discuss the informations gathered.

4.1 Continuous model

The first simulation will be performed on the continuous model, which has qualitative means only. It will tell us if the model responds as we would expect, so we can proceede with discretizing the subsystem which implements the *Three-Loop* algorithm. If we spot anything suspicious here, it means there have been made some mistakes in the modelization process and hence the model must be revisited. The Simulink model is shown in Figure 12. By issuing a commanded acceleration input signal a_c, the corresponding ac-

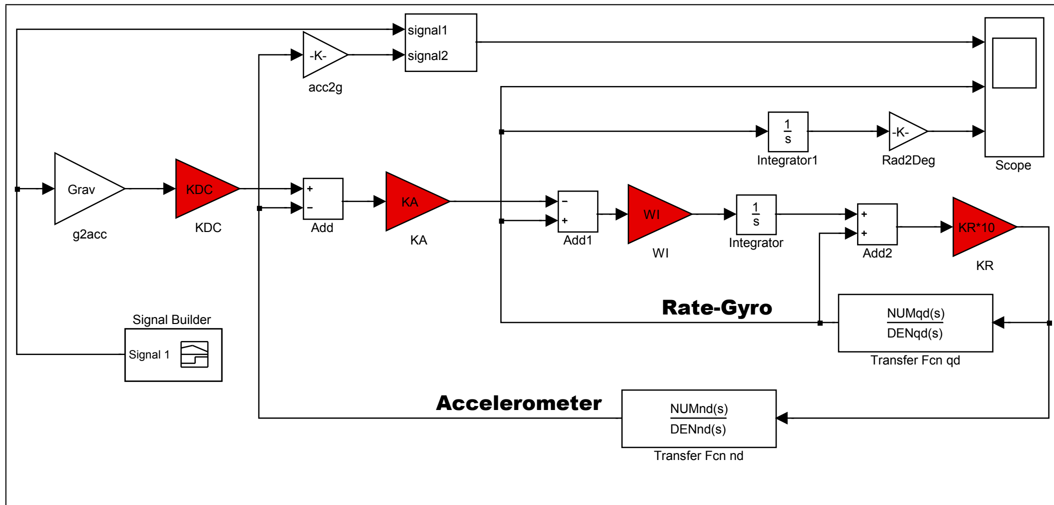
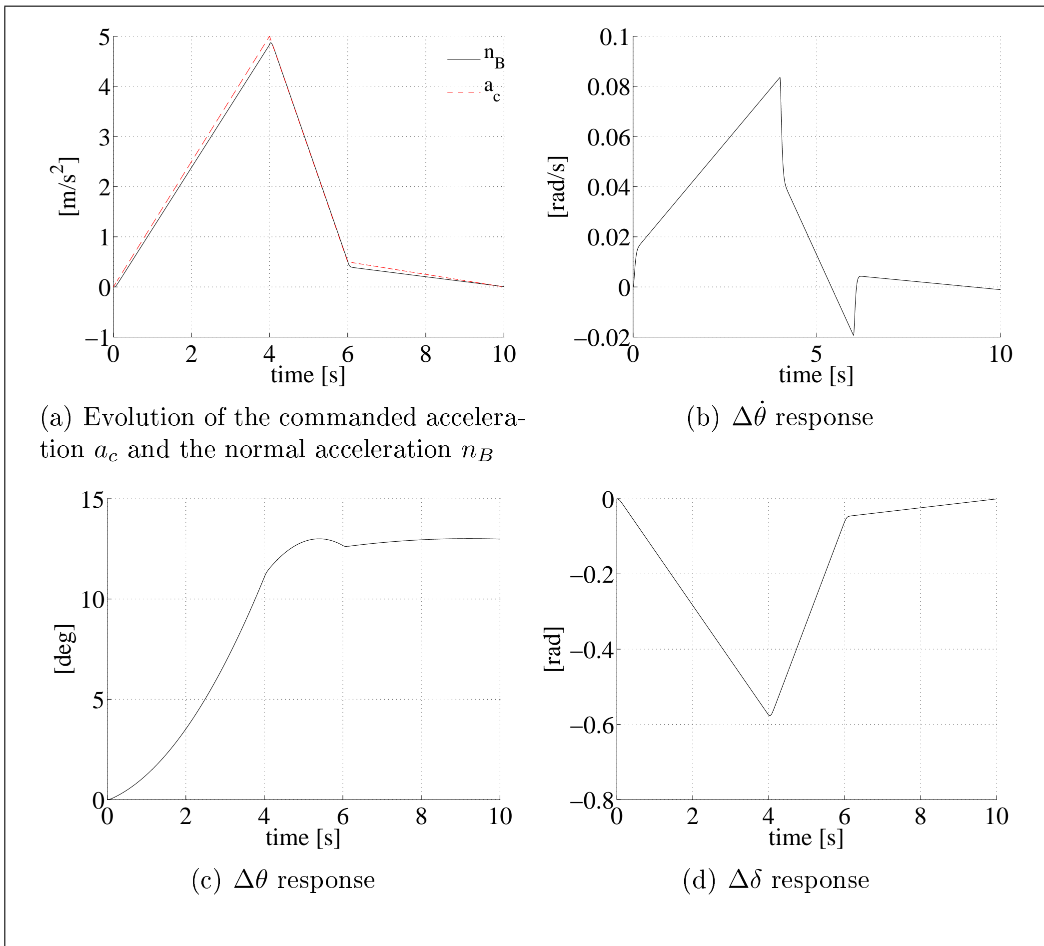


Figure 12: Simulink schema of the system.

celeration normal to the missile axis n_B which causes the desired trajectory change is showed in Figure 13.

As we can see, the acceleration n_B overlaps almost perfectly[7] the signal a_c, confirming the sinthesys made in the previous chapter. We can also notice the evolutions of the input and output feedback signals: $\Delta\dot{\theta}$ smoothly increases to generate an adequate F_N which keeps the missile on track, while

[7] A small delay is present because of the dynamics of the loop.



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 13: Follow-up of overall evolution of the signals, continuous model.

$\Delta\delta$ denotes the angle of the tail fin needed to modify the angular velocity, and therefore M, and perform the trajectory change.

4.2 Discretized model

Since the continuous model simulation was good, we can now proceede with a more refined model, which will also take into consideration the compound nature of the system. For that reason, we are going to discretize part of the model representing the Three-Loop algorithm, while keeping unchanged the missile model. As you can guess from Figure 12, the entire Three-Loop algorithm is composed of simple algebraic operations, and a continuous integrator. Hence, according to [4], a first approximation is to insert a `Zero-Order Hold` after the two transfer functions output, and convert the continuous integrator into a discrete one. After these improvements, the model obtained

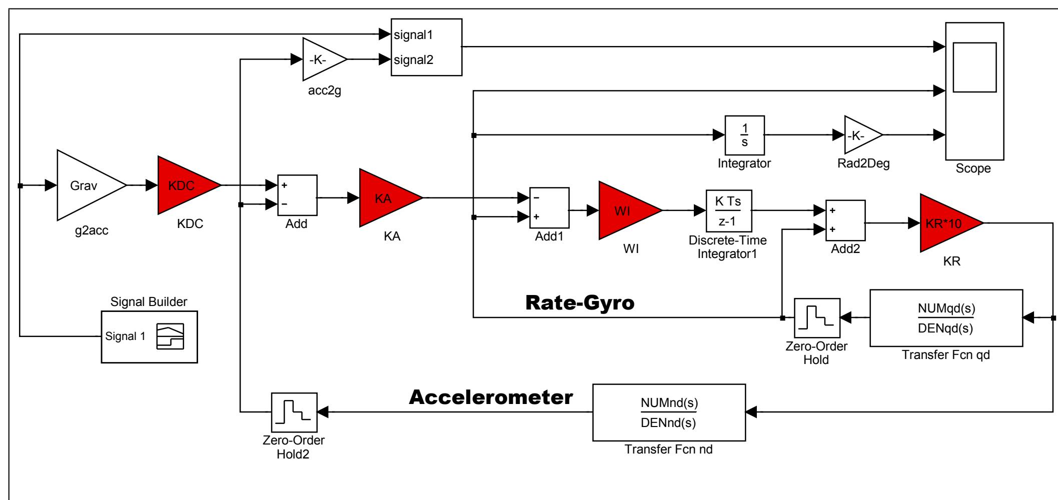
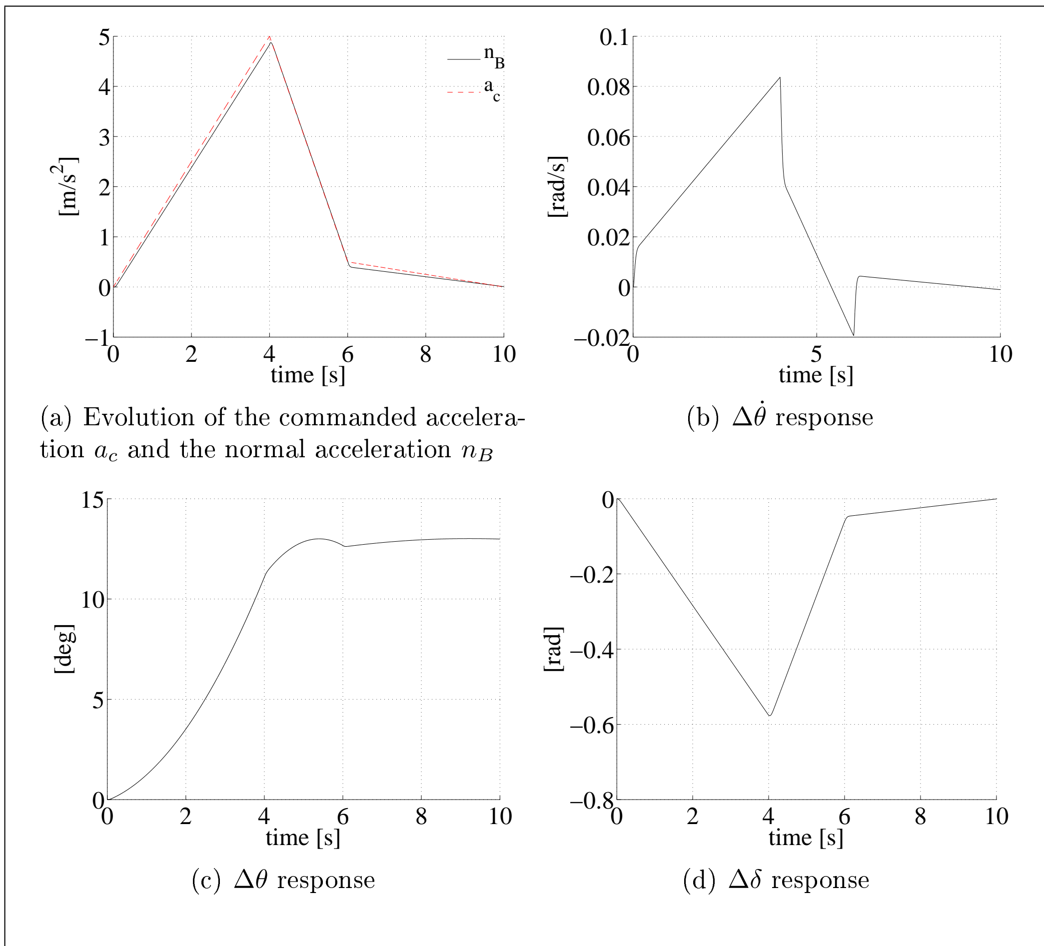


Figure 14: Simulink schema of the discretized system.

is shown in Figure 14. Running a simulation with that model will produce the results exposed in Figure 15.



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 15: Overall evolution of the signals, with discretized *Three-Loo Algorithm.*

As we just saw, the overall responses of the signals are almost the same of the continuous case, therefore we need another level of approximation to notice something different.

4.3 Discretized model - explicit loop delay

Since the previous simulations were not close enough to the real model, we will now consider *how* the *Three-Loop Algorithm* works inside an electronic device:

- at first, it collects all the inputs neeeded;
- then, it performs the computations and obtains the result;
- finally, the result is made available at the chip output;

All those operations are not istantaneous, they will take some time to be perfomed indeed. Hence, a non neglibile delay between the input acquisition and output has to be introduced in the model. We need an upper bound

Figure 16: Simulink schema of the complete system with delay block.

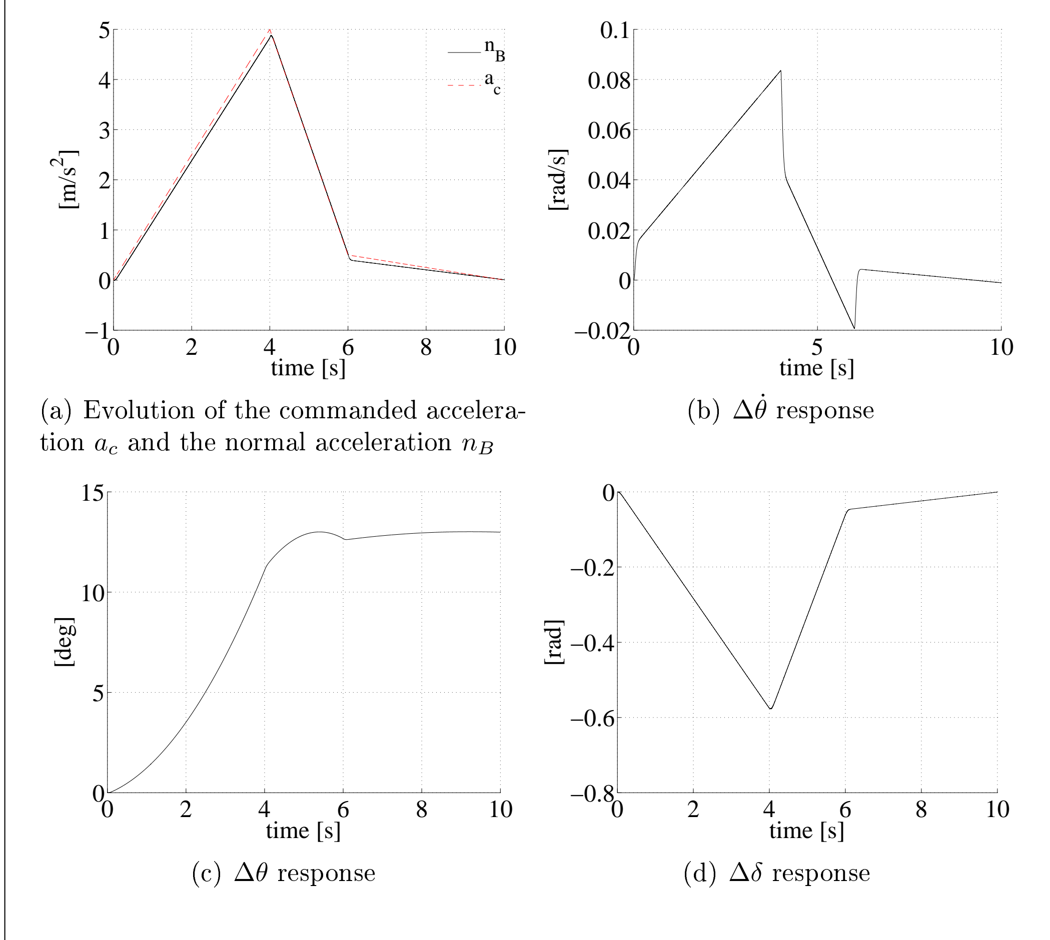
for the maximum execution time of the loop, easily given by the number of operations performed by the algorithm multiplied by T_{CY}, the instruction execution time. Assuming:

- operational frequency of $40MHz \longrightarrow T_{CY} = 100ns$;
- presence of 15 algebraic operations inside the loop;
- an average of 8 microcontroller instructions needed for each operation, which sums up the retrival of the operands from their temporary location, the actual operation call, and the storage of the result in an other temporary memory address;
- the ADC module acquisition and conversion time, which lasts about $30\mu s$;

we obtain

$$T_{LOOP_{UpperBound}} = 15 \cdot 8 \cdot 100 \cdot 10^{-9} + 4 \cdot 30 \cdot 10^{-6} = 132 \mu s \tag{4.1}$$

To play safe, we further increase the $T_{LOOP_{UpperBound}}$ to $200\mu s$, bringing the results shown in Figure 17.



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 17: Overall evolution of the signals, with discretized *Three-Loo Algorithm.*

If, at a first sight, n_B response seems unchanged, a closer look will reveal an interesting behavior, far different from the previous two simulations made. In fact, the cyan respose in Figure 18 now resembles a sawtooth wave overlapping n_B response seen in the previous section. This behavior is closer to the real case than the other two, because it nicely describes how a delayed, discretized controller acts: since its output is made available to the outside after a certain amount of time, during which the input has evolved,

its response shows an oscillating component due to the efforts the controller makes to compensate its previous action.

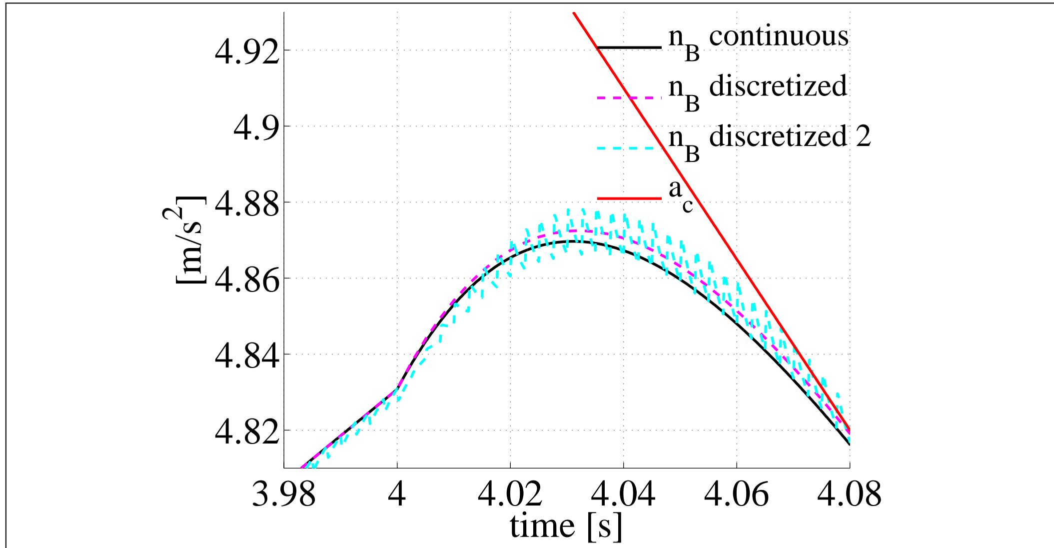


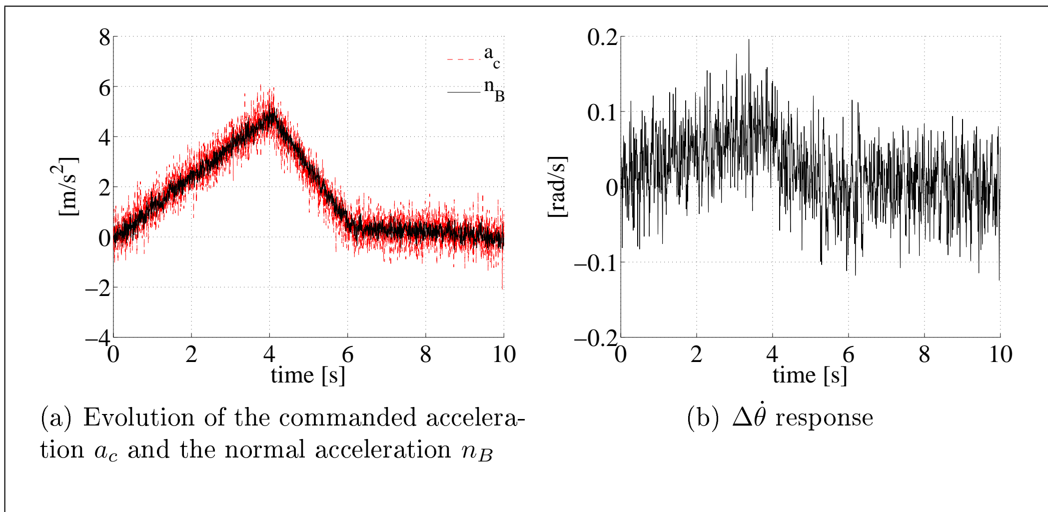
Figure 18: Overall responses of n_B for the three simulations made.

4.4 Noise disturbance effects

Until now, we saw how the system evolves in ideal conditions. Here, we are interested about how the model behaves when there is a source of electrical noise applied to one of its inputs. We are asking ourselves: will the controller be able to perform its duty even in presence of noisy signals? How the model will react to such stimulus? Which gimmicks may be used to reduce the effects of the noise?
All the results exposed hereafter were made with the discretized model with explicit delay block, and an $AWGN$ [5] signal X applied. Due to the nature of the signals applied at the microcontroller's pins, in the $[0, 5V]$ range, we will consider first a very noisy source of disturbance of $\pm 10\%$ the maximum value, that is $\pm 0.5V$, so $X \sim N(0, 0.25)$. Then, a more sensible noise disturbance $X \sim N(0, 0.0025)$ will be applied.

4.4.1 Noise disturbance $\sim N(0, 0.25)$ applied to a_c

In this section, the system will be supplied with the usual commanded acceleration overlapped with the noisy signal X previously defined. From

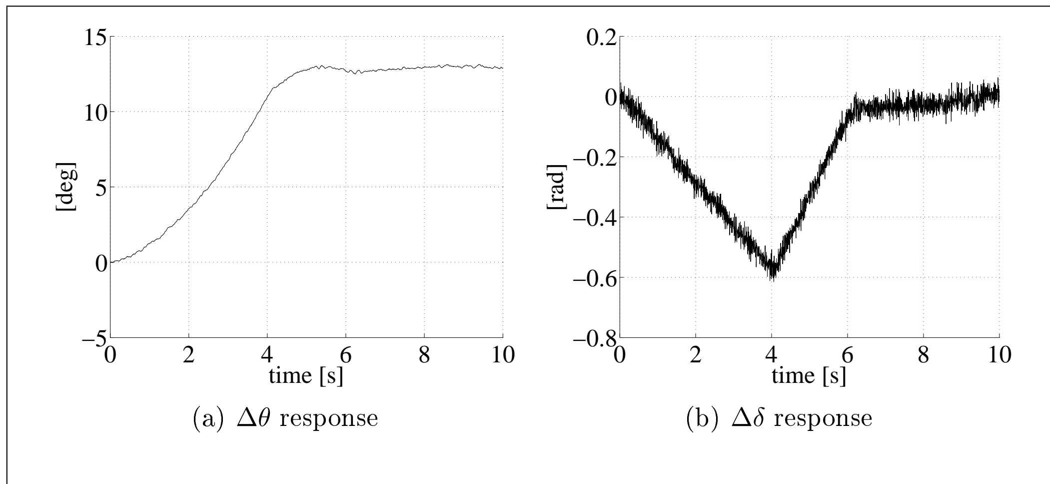


(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

Figure 19: Overall evolution of the signals characterizing the algorithm, with noise applied to a_c.

the results exposed in Figure 25 and Figure 26, n_B still follows a_c evolution even though it is edgy, while $\Delta\theta$ response is almost identical to the ideal simulations made.

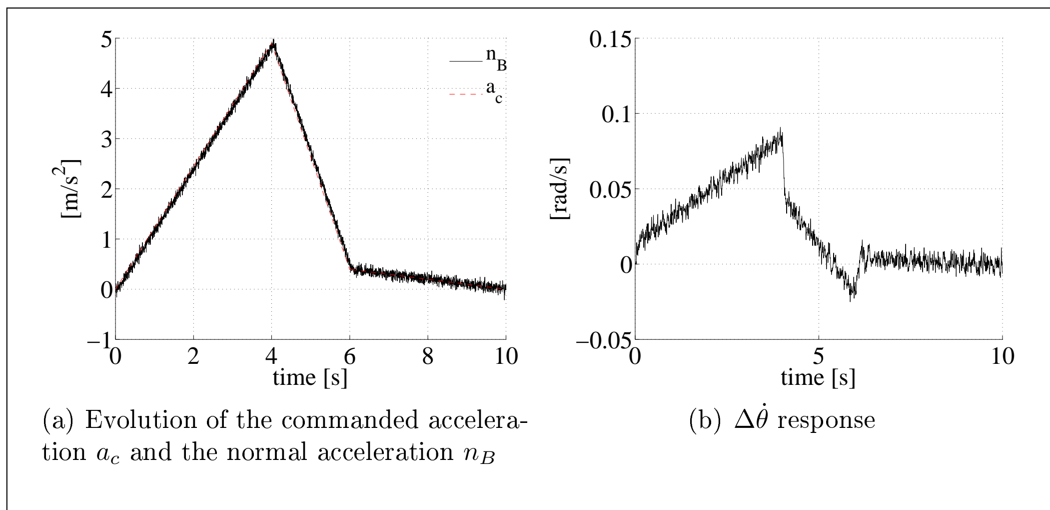


(a) $\Delta\theta$ response (b) $\Delta\delta$ response

Figure 20: Overall evolution of the signals characterizing the algorithm, with noise applied to a_c, continued.

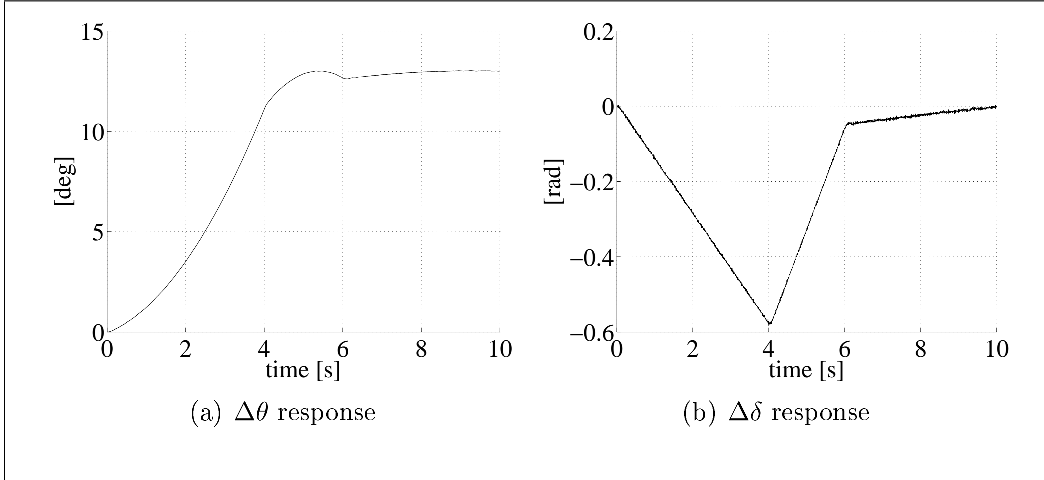
4.4.2 Noise disturbance $\sim N(0, 0.25)$ applied to n_B

Now we will see what happens when the commanded acceleration evolves smoothly, while n_B is affected of the source of noise X aforementioned.



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B (b) $\Delta\dot{\theta}$ response

Figure 21: Overall evolution of the signals characterizing the algorithm, with noise applied in n_B.



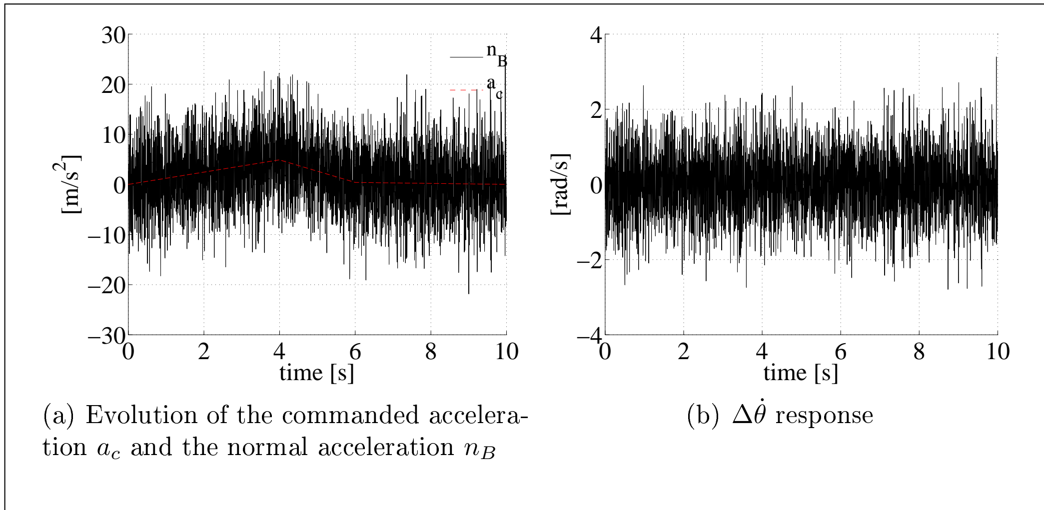
(a) $\Delta\theta$ response (b) $\Delta\delta$ response

Figure 22: Overall evolution of the signals characterizing the algorithm, with noise applied in n_B, continued.

As seen in Figure 21 and Figure 21, n_B exhibits a marked noise rejection property, making the entire controller behaving seamlessly.

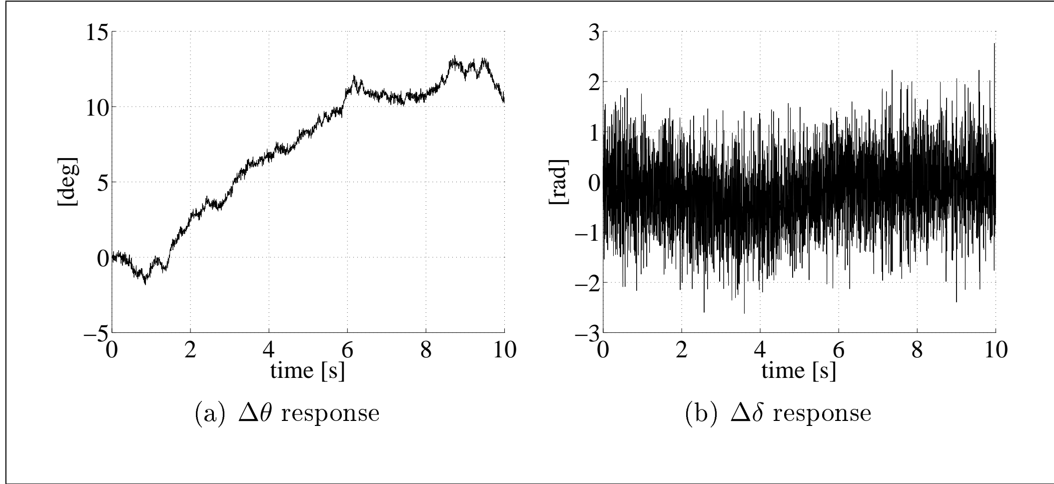
4.4.3 Noise disturbance $\sim N(0, 0.25)$ applied to $\Delta\dot{\theta}$

Here we added the $AWGN$ signal X to $\Delta\dot{\theta}$, obtaining the results below:



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B (b) $\Delta\dot{\theta}$ response

Figure 23: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\dot{\theta}$.



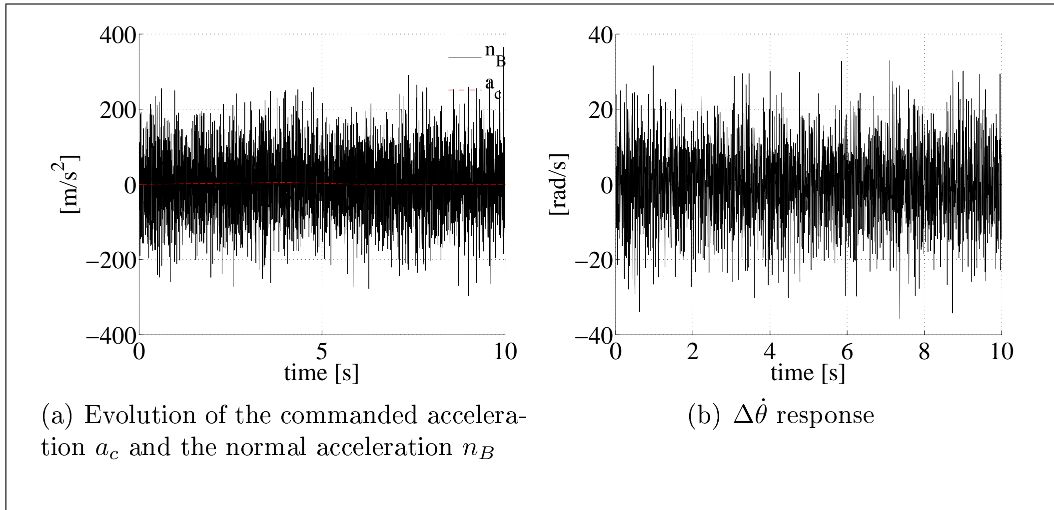
(a) $\Delta\theta$ response (b) $\Delta\delta$ response

Figure 24: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\dot{\theta}$, continued.

Due to the small range of values assumed by $\Delta\dot{\theta}$, the effects of the noise are way more noticeable than the two previous simulation. That means extra care must be taken when designing the connection carrying $\Delta\dot{\theta}$ signal, otherwise a source of noise will lessen the overall missile's performances.

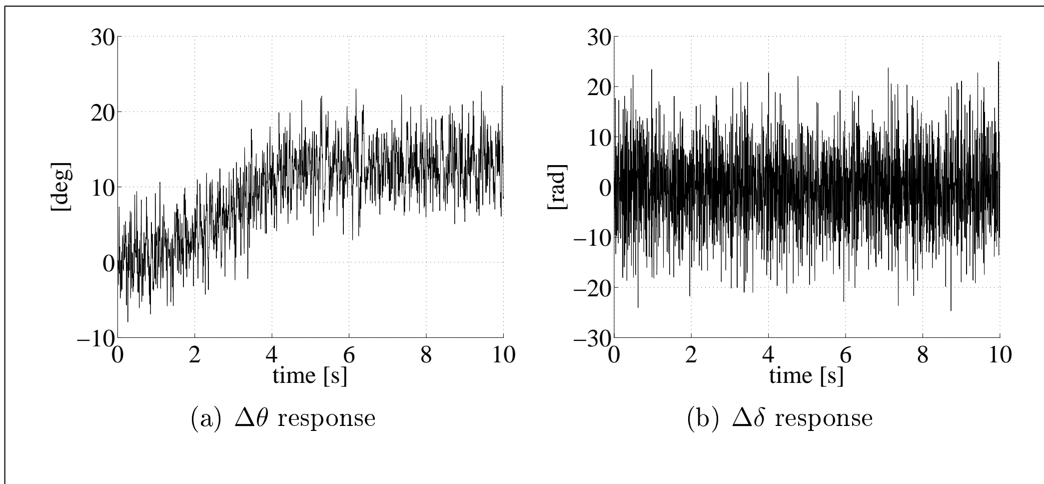
4.4.4 Noise disturbance $\sim N(0, 0.25)$ applied to $\Delta\delta$

Here we add the $AWGN$ signal X to$\Delta\delta$, obtaining the results shown below:



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B (b) $\Delta\dot{\theta}$ response

Figure 25: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\delta$.



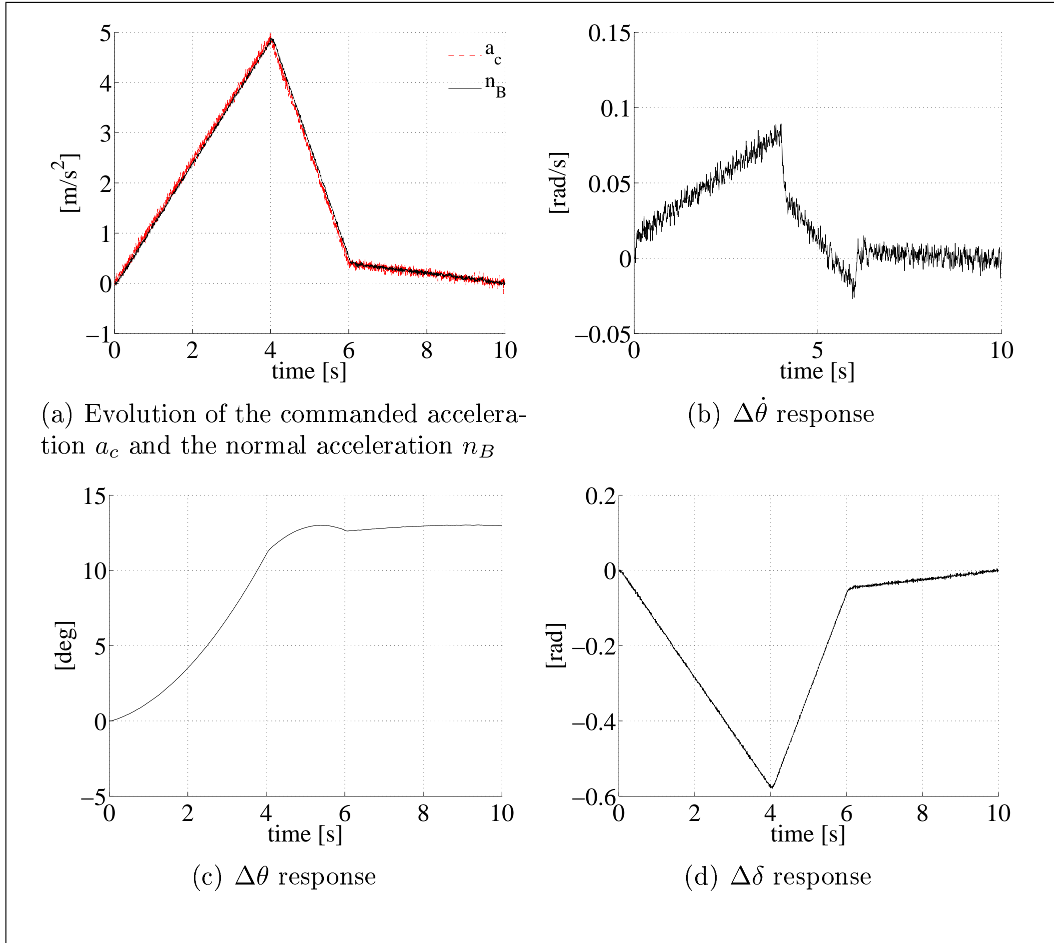
(a) $\Delta\theta$ response (b) $\Delta\delta$ response

Figure 26: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\delta$, continued.

The same deduction made in the previous section applies here too: the nature of the $\Delta\delta$ implies careful design in order to make it free from noise disturbances.

4.4.5 Noise disturbance $\sim N(0, 0.0025)$ applied to a_c

Since a ±10% noise a highly unlikely case, starting fom now, until the end of this chapter, we will consider a more realistic case of ±1% of noise signal. Let's assume the commanded acceleration a_c being polluted by an AWGN signal with zero mean, and $\sigma^2 = 0.0025$: the responses obtained are shown in the figures below.



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

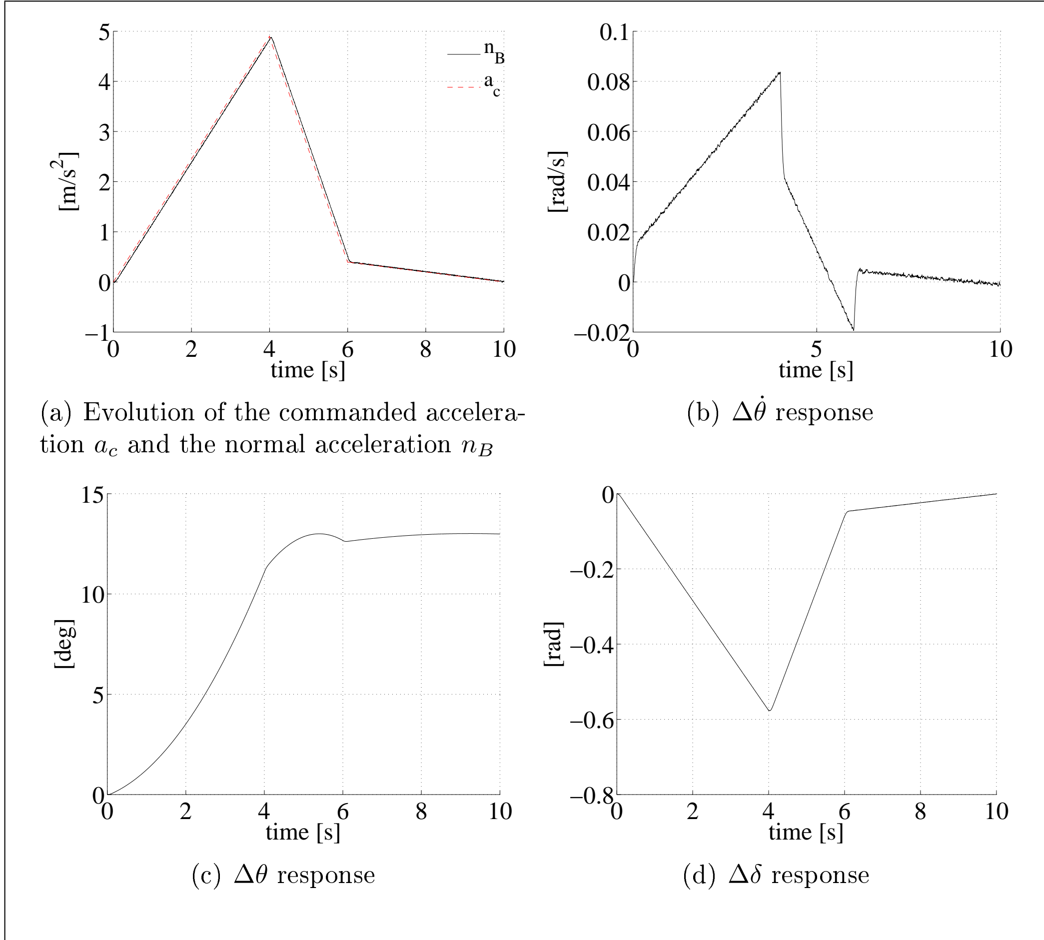
(d) $\Delta\delta$ response

Figure 27: Overall evolution of the signals characterizing the algorithm, with noise applied to a_c.

With such realistic noise, the whole system exhibits a good disturbance rejection; the feedback signal $\Delta\delta$ evolves much more smoothly compared with the ±10% case and, consequentely,the acceleration signal n_B is much smoother too which means that globally, the physical model is less strained by the noise applied.

4.4.6 Noise disturbance $\sim N(0, 0.0025)$ applied to n_B

We already noticed how good is the whole system at rejecting a noisy source applied to n_B, so the results we expect will be most likely better than the $\pm 10\%$ case. That's exactly what we expected: both the acceleration n_B and



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

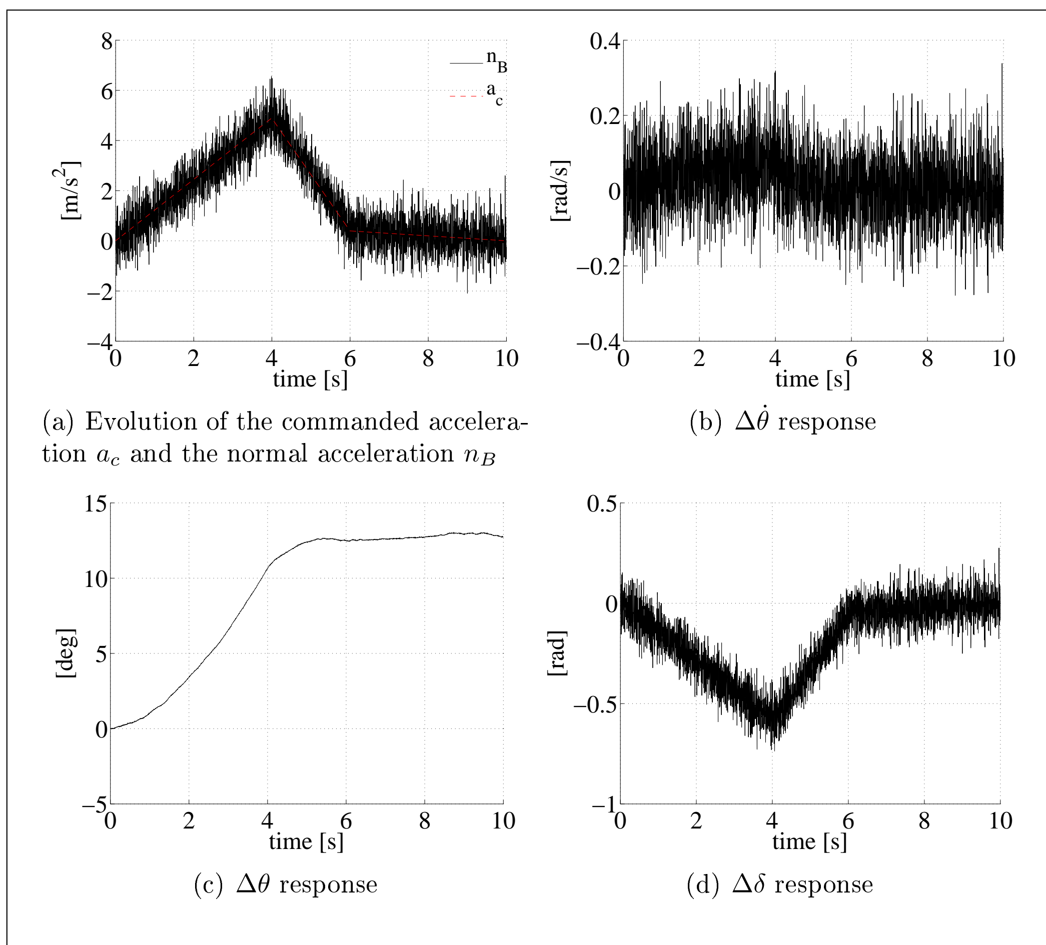
(d) $\Delta\delta$ response

Figure 28: Overall evolution of the signals characterizing the algorithm, with noise applied to n_B.

the feedback signal $\Delta\delta$ show a little ripple due to the noise entered in the control loop.

4.4.7 Noise disturbance $\sim N(0, 0.0025)$ applied to $\Delta\dot{\theta}$

Let us see now how $\Delta\dot{\theta}$ evolves with such noise. As we just saw, the evolution



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

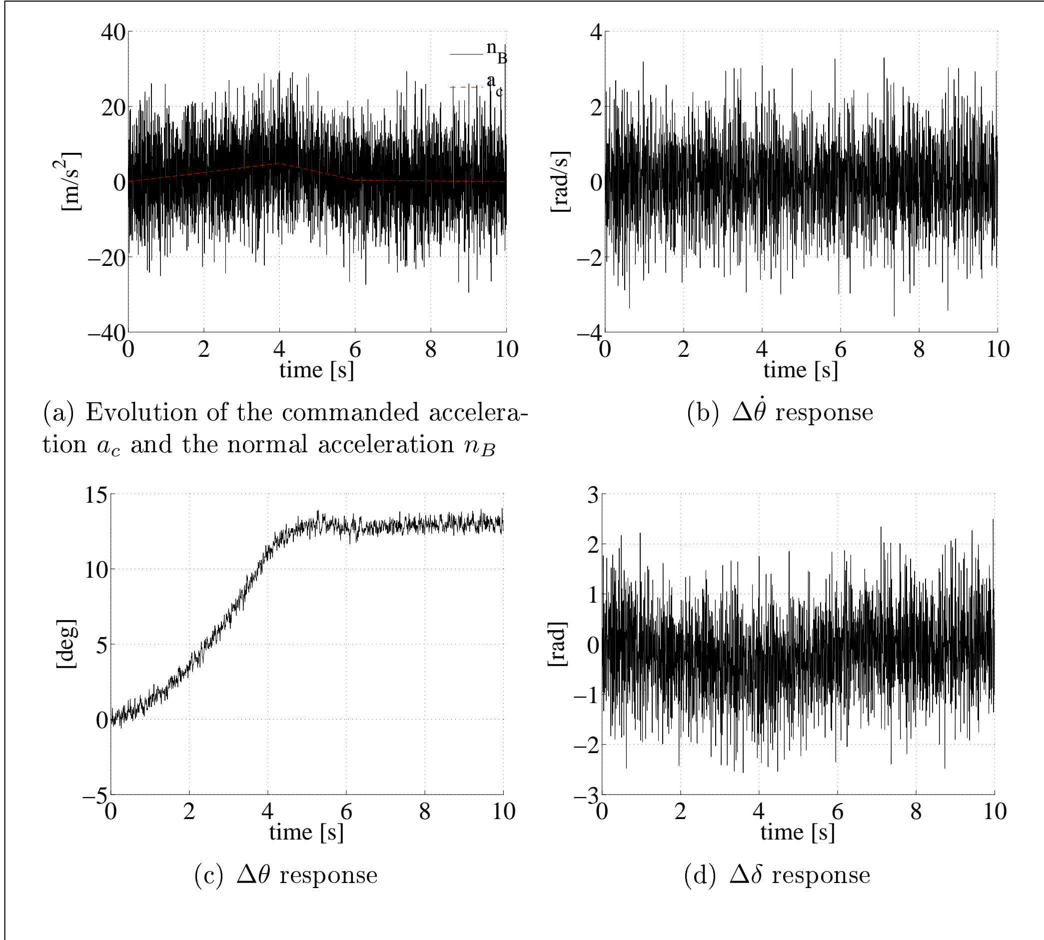
Figure 29: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\dot{\theta}$.

of the signals is greatly improved compared with the $\pm 10\%$ case but still, they are not good enough for our purposes. Therefore, extra care must be taken to avoid noise disturbance interfering with the signal $\Delta\dot{\theta}$[8].

[8] i.e. by placing the controller next to the gyro and using shielded cables for the connections.

4.4.8 Noise disturbance $\sim N(0, 0.0025)$ applied to $\Delta\delta$

We are now interested to see how the signals evolves when the feeedback one gets polluted by AWGN signal. In the $\pm 10\%$ case the model was completely unmanageable; let's see now what happens. The only visible improvement



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

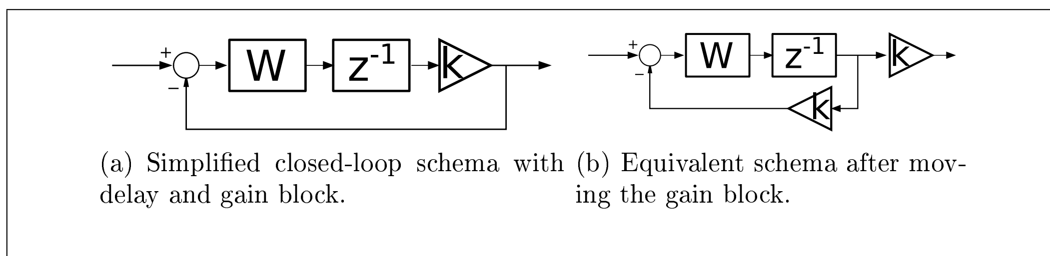
(d) $\Delta\delta$ response

Figure 30: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\delta$.

made is a somewhat continuous $\Delta\theta$ evolution, while the others are still too much noisy to perform any kind of control. Again, particular care must be taken to reduce the possibilities for the noise to mess with $\Delta\theta$.

4.4.9 Noise disturbance applied to $\Delta\delta$, take two

Since the previous simulation was unacceptable, we need to find an alternate way to make the noise applied in $\Delta\delta$ more tolerable by the system. If we look at the block models in Figure 9 and Figure 16, we notice $\Delta\delta$ coming from a delay block preceded by the scalar multiplication $KR \cdot 10$. If we stop for a moment in that scalar multiplication, we would notice that $KR \cdot 10 = 73.657 \cdot 10^{-3}$, so the value of $\Delta\delta$ gets delayed and actually *scaled* by a factor of ~ 13.5 !
If we could find a way to remove that operation, the signal-to-noise ratio will be improved enough to permit the engineers to design connections between the system and the controller able to deal with such noises.
Removal of that block is of course not possible because it would change the closed-loop response, however we can trick the model as follows. If we consider the procedure of scaling a signal and then its delaying, we can assert that it is equivalent of delaying the signal first, and then perform the scaling operation.
Therefore, it is legitimate to swap the order of the two blocks. After that swap, we are now able to perform two tricks on the simplified scheme blocks, shown in the figure here below:

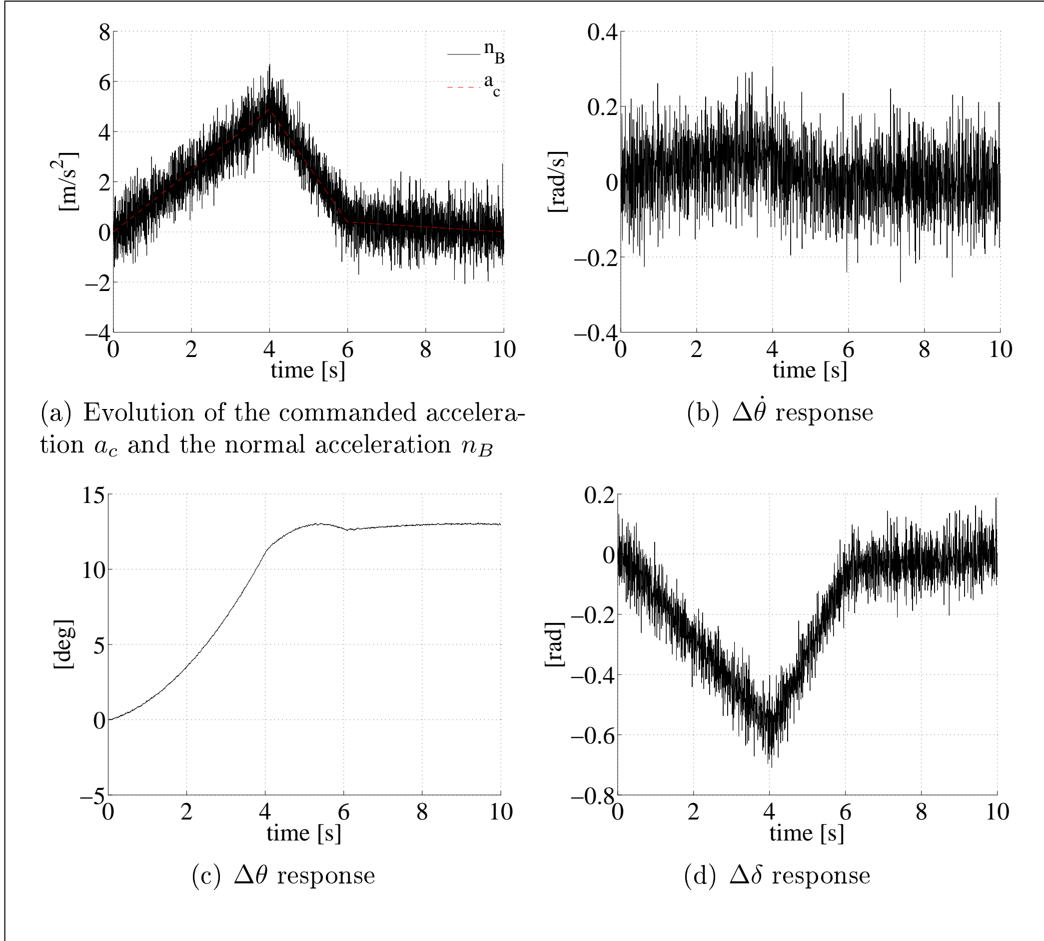


(a) Simplified closed-loop schema with delay and gain block.

(b) Equivalent schema after moving the gain block.

Figure 31: Gain block repositioning: before and after.

Applying this gimmick to the discretized model, and repeating the simulations, leaded us to the following plots:



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 32: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\delta$ and block gain repositioned.

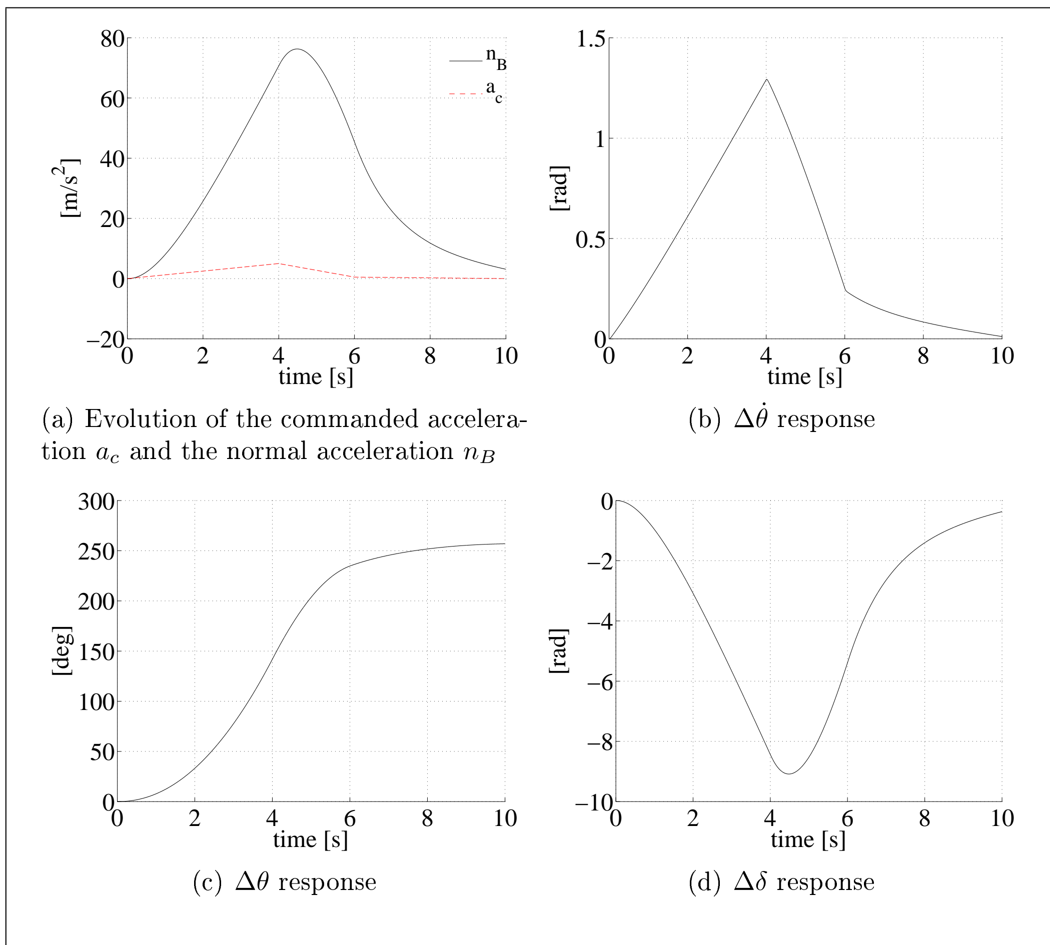
That's a huge improvement compared with our last simulation, and even though the signals are higly disturbed by the noise applied, the overall signal-to-noise ratio is much great than the previous one, hence it will be easier to come up with a solution to protect from the noises threatening $\Delta\delta$.

4.5 Faulty conditions analysis

We want now to investigate how the system reacts under some faulty condition, like a missing feedback signal due to the physical link breakage. We omitted of course the cases of a_c signal interrupted because the missile won't move at all, and also the case of $\Delta\delta$ absence because it means that $\Delta\delta$ signal coming out from the controller doesn't reach the actuators and therefore the missile won't start.

4.5.1 Absence of n_B

Suppose the connection between the accelerometer and the controller for some reason breaks, how the entire system will react? As seen in Figure 33,



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

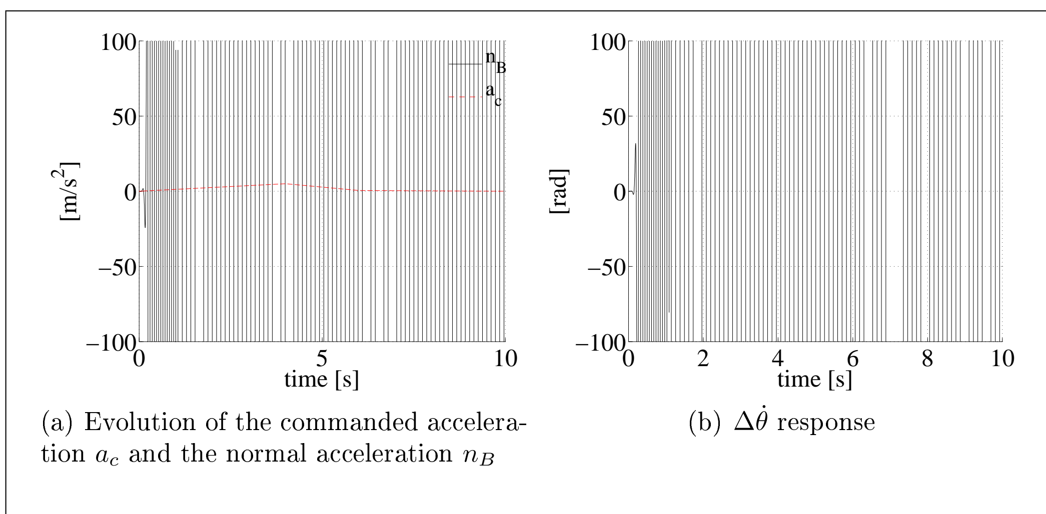
Figure 33: Overall evolution of the signals characterizing the algorithm, without n_B.

the absence of feedback from n_B makes the controller act boldly in the vain

attempt to compensate the gap between a_c and the supplied measure of n_B, which is zero in this simulation. This creates a feedback action one order of magnitude greater that the standar case, causing the missile to undergo accelerations for which it wasn't designed for, with the risk of damaging itself. It is also interesting to note how the overall responses of the signals resemble the standard ones[9]; that means the two inner control loops are responsible of the signals evolution, while n_B of the outer loop controls the magnitude of the other quantities.

4.5.2 Absence of $\Delta\dot{\theta}$

With the deductions made previously, we are now interested on seeing how the system reacts when the two feedbacks from $\Delta\dot{\theta}$ get interrupted.

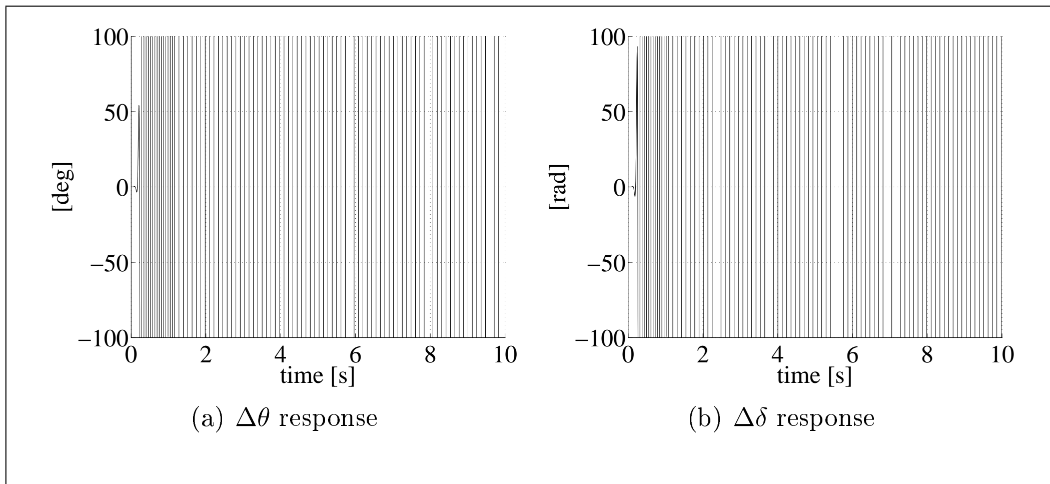


(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

Figure 34: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\dot{\theta}$.

[9]Neglecting of course the higher mangitude of the latter.



(a) $\Delta\theta$ response

(b) $\Delta\delta$ response

Figure 35: Overall evolution of the signals characterizing the algorithm, without $\Delta\dot{\theta}$.

As we can see from Figure 34 and 43, the responses of the signals without feeedback action from $\Delta\dot{\theta}$ are completely unstable and therefore the system cannot be controlled at all.

4.6 Moving toward a Co-Simulated approach

Until now we made out simulations based upon a continuous missile model and a discretized representation of the microcontroller. Even though we gathered a lot of interesting results from those simulations, we do also realize the microcontroller discretization is still too much approximated and does not capture the most interesting behavior the microcontroller exhibit. One example for all: the ADC capture and decode process.
In the discretized representation of the controller, the analog inputs are collected and processed istantaneously; in the real world, every ADC read and conversion is performed sequentially, and requires some time to be completed. Furthermore, every conversion introduce the so called *granular error* $e_q = \frac{\Delta}{2}$ which could be problematic to deal with[10]. For this reason, and others that will come, we need to switch to a co-simulated approach.

[10] Δ is the so called *quantization step*, given by $\frac{2v_{sat}}{2^{bit}}$[5]

5 Three-Loop Co-Simulation

As we saw in the previous chapter, modeling the Three-Loop algoritm by means of pure mathematical expressions doesn't really give us a complete idea about how the microcontroller implementing that algorithm will interact with the missile model. Due to the nature of matlab in fact, it is impossible to easily describe the following phenomena all together:

- the finite nature of the resources made available by the hardware provided for the $PIC18F4620$ microcontroller;
- the delays and quantization errors made when converting an analog signal to its digital counterpart;
- the delay between the inputs being collected, and the instant when the output is made available to the system;
- the *finite arithmetic* available on the chip that is, 8-bit operations;
- the absence of floating point arithmetic instructions, which must be emulated through fixed point instruction routines, with consequent increase of the main loop execution time.

This is not even the worst case scenario: if you think about the presence of the so called *interrupt service routines*[11] (ISR), used to free the microcontroller of the time-expensive *busy wait* paradigm, you will realize that matlab alone is no longer able to provide accurate simulations. This is why we are now going to move into the *Co-Simulation* world, where the firmware will be executed as if it would be inside a real controller and, at the same time, it interacts with the matlab models representing the missile components.
Before start going any further, we need to write down the implementation of the Three-Loop algorithm which requires, in turn, to know exactly which modules of the microcontroller are needed to accomplish our purpose.

5.1 Algorithm characterization

The algorithm itself is not complicated: looking at Figure 16, we notice the presence of simple arithmetic operations such as sums and multiplications, and just one discrete integrator.
However, due to the nature of the microcontroller in use, we need to set up extra modules needed to interact with the surrounding world; such operation will make the complete program more complex as we would ever expect with a pure matlab simulation.

[11] Invoked for example when an ADC conversion has been completed.

5.1.1 Input module

In order to process the inputs coming from the outside, we have to retrieve the signals through the I/O pins the chip provides. Since the nature of the signals in use is analog, we cannot simply rely on the digital I/O pins provided.
Instead, we must turn on and set up correctly the ADC module [1]. Then, for each sample we need to retrieve, we have to notify the module to start a new conversion, and wait for the result.
The registers involved in the setup and functioning of the ADC module are[1]:

- $ADCON0$, the A/D control register 0: an 8-bit register, which controls the module operational status:
  - bit 7-6: unimplemented;
  - bit 5-2: *CHS<3:0>* analog channel select bits. Tells the microcontroller which of the pins connected with the module is currently being used;
  - bit 1: $GO/\overline{DONE}$ status bit;
  - bit 0: $ADON$ enable bit.
- $ADCON1$, the A/D control register 1:an 8-bit register, which controls the module port usage:
  - bit 7-6: unimplemented;
  - bit 5: $VCFG1$, the voltage reference configuration bit. Tells whether the module must use VSS ($VCFG1 = 0$) or an external negative reference ($VCFG1 = 1$);
  - bit 4: $VCFG0$, the voltage reference configuration bit. Tells whether the module must use VDD ($VCFG0 = 0$) or an external positive reference ($VCFG0 = 1$);
  - bit 3-0: A/D Port Configuration bits control, tells the module whic pins are configured ad analog inputs.
- $ADCON2$, the A/D control register 2: an 8-bit register, which controls the module conversion modes:
  - bit 7: $ADFM$, tells whether the result stored is left or right justified;
  - bit 6: unimplemented;

- bit 5-3: $ACQT<2:0>$, provides a range of acquisition times multiple of T_{AD}[12]
- bit 2-0: $ADCS<2:0>$, provides a range of conversion clock select bits tied with the frequency of the oscillator.

- $ADRESH$ and $ADRESL$: the registers holding the high and low part of the conversion result.

As you can see, the number of parameters for configuring and making the module just work is pretty high however, with help of the manufacturer data sheet [1], its configuration will be easy. From the table[2], $T_{AD_{min}}$ for our microcontroller equals to $0,7\mu s$. Since it must be verified the following equations

$$T_{AD} = ADCS<2:0>\cdot T_{OSC} \geq T_{AD_{MIN}} \tag{5.1}$$

we can easily determine $ADCS<2:0>$. Assuming an operational frequency of $40MHz$, hence $T_{OSC} = 25ms$, we get:

$$ADCS<2:0> = \left\lceil \frac{T_{AD}}{T_{OSC}} \right\rceil = 7 \tag{5.2}$$

Since 7 is not a valid multiple of T_{osc}, we choose the nearest, increasing one, which is $ADCS<2:0> = 8$. Given the need of three analog inputs, that is a_c, n_B and $\Delta\theta$, and according to what we said before about the constraints, the vaules of those registers will be

$$\begin{cases} \text{ADCON0} = \text{0b00000001} \\ \text{ADCON1} = \text{0b00001100} \\ \text{ADCON2} = \text{0b00101001} \end{cases} \tag{5.3}$$

Of course $ADCON0$ value is not fixed, because it changes accordingly with the ping being multiplexed in the ADC module.
Once set up the module, we must follow these steps to perform an A/D conversion:

- select the desired pin by means of $CHS<3:0>$;
- wait T_{ACQ};
- start the conversion by setting $GO/\overline{DONE}$ bit;
- loop until $GO/\overline{DONE}$ gets cleared, meaning the conversion ended;
- read the value from $ADRESH$ and $ADRESL$;

[12] T_{AD} is the amount of time needed by the module to convert one bit.

5.1.2 Limited input values

Inside matlab, you can apply whatever commanded acceleration a_C, and the simulation, be it correct or wrong, will be always performed. This is not true in the real world case of course: if you apply a signal with amplitude of say, 30 Volts, the microcontroller will blow up for sure.
Therefore we need to scale and traslate the inputs, if necessary, in order to fit the microcontroller admissible range for its inputs, which is in our case $[0,5]Volt$ and then, in the firmware, convert back the values to their original ones. Of course if a signal is too small to be converted, i.e. his magnitude si equal to the ADC *quantization step* $= \frac{5}{2^{10}} = 4.88 \cdot 10^{-3}$, the signal must be expanded to get a better resolution[13]. By inspecting Figure 17 and assuming a_c in the $[0, 5V]$ range:

- a_c does not need any conversion since it is already in the range granted from the PIC hardware. However, inside the PIC, the value will be converted by the ADC module in a integer value between $[0, 1023]$ so, to obtain the original value back, we have to multiply the result of the conversion by 0.004875;

- n_B must be scaled to fit the aforementioned range. Since its values are in the $[0, 50V]$ range, it is sufficient to perform a simple division by a 10 factor. Inside the microcontroller, to restore its original value we have to multiply the converted result by 0.04875;

- $\Delta\theta$ belongs to the $[-0.03, 0.1V]$ range, so to fit the PIC's ADC range and get a better conversion, the original signal must be translated by adding the constant 0.03 and then multiplied by 38.46. Once inside hte PIC, the converted value has to be multiplied by 0.000127077 and then subtract 0.03 to retrieve the original value.

5.1.3 Output module

Once all the computations are preformed, the microcontroller has to forward its analog result to the outer world, right into the missile model. There is a little problem however: the PIC family doesn't have any analog output pin, so our problem now is: how is it possible to send the result outside?
One solution could be considering buying an expensive module expansion called *dsPIC* made exactly for this purpose; however this solution is overkilling because that module has *a lot* of features that will never be used in our application.

[13]But, of course, this will expand the noisy component overlapping the signal too.

An other solution could be to use eight or ten pins of the microcontroller, and send the result out in parallel. This solution isn't optimal because wastes a lot of peripherals that could be used for other purposes.
There is a way out however: since every PIC has a PWM module we could use the result, appropriately converted, to modulate a PWM waweform, thus using just one pin. The PWM waveform is sent to the missile model and, before being processed, its duty cycle value will be converted to the original[14] microcontroller result.

As we previously saw with the ADC module, PWM is higly configurable too but this time, to get things more complicated, it actually belongs to the CCP[15] module. The register needed to set up and use PWM module are:

- $CCP1CON$, the CCP1 control register: an 8-bit register which controls the module behavior:
  - bit 7-6: unused;
  - bit 5-4: they represent the two LSB of the 10-bit PWM duty-cycle, if the module works as PWM, or undefinded otherwise;
  - bit 3-0: $CCP1M<3{:}0>$ are the CCP Module select bits;
- $CCPR1L$, the 8-bit register holding the 8 most signicative bits of the duty cycle value;
- $PR2$, the 8-bit register which determine the duration of the PWM period T_{PWM};
- $T2CON$, the 8-bit Timer2 control register used by the PWM module to produce the final waveform, where:
  - bit 7-3: don't take part in the PWM calculus, hence are undefined;
  - bit 2: $TMR2ON$, enables the Timer2 module;
  - bit 1-0: $T2CKPS<1{:}0>$, reprensents the Timer2 Clock prescale select bits;

Now we can finally see how all those registers are tied together to produce the desired T_{PWM} and T_{DUTY}. According with[3], the formulas are

$$\begin{cases} T_{PWM} & = [PR2 + 1] \cdot 4 \cdot T_{OSC} \cdot T2CKPS<1{:}0> \\ T_{DUTY} & = (CCPR1L : CCP1CON<5{:}4>) \cdot \\ & \quad T_{OSC} \cdot T2CKPS<1{:}0> \end{cases} \tag{5.4}$$

[14] There are quantization errors of course in play.
[15] Capture/ Compare/PWM Module.

If we want to use the module at its maximum resolution, that is $2^{10} = 1024$ levels, register $PR2$ must be set ot its maximum value too, which is $2^8 = 256$. There is only one degree of freedom in the equations, given by $T2CKPS<1:0>$. By tweaking it, T_{PWM} minimum and maximum value will be

$$\begin{cases} T_{PWM_{min}} = 25,6\mu s & \text{if } T2CKPS<1:0> = 1 \\ T_{PWM_{max}} = 409,6\mu s & \text{if } T2CKPS<1:0> = 16 \end{cases} \tag{5.5}$$

Which PWM period is the most suitable for our application?
If we consider the approximations made in Section 4.2, where we estimated the loop duration about $200\mu s$, we safely choose $T_{PWM} = T_{PWM_{min}}$, and the corresponding register values are

$$\begin{cases} CCP1CON = 0b00001100 \\ PR2 = 0b11111111 \\ T2CON = 0b00000111 \end{cases} \tag{5.6}$$

5.1.4 Loop execution time estimation

As we saw in Figure 16, the algorithm must implement the discrete integrator

$$y_k = y_{k-1} + u_k \cdot T_L \tag{5.7}$$

T_L is the *loop execution time*, and its value must be carefully calculated, in order to get a reliable result.
The question now is: how could we exactly measure the loop execution time? Thanks to the particular arcitecture of the PIC products,

> [...]all single-word instructions are executed in a single instruction cycle, unless a conditional test is true or the program counter is changed as a result of the instruction. In these cases, the execution takes two instruction cycles.

Therefore, we just need to count the assembly instructions contained in the main loop, multiplied with their number of instruction cycles, and then multiply the final value by T_{CY}.
There still is however a flaw with that approach, originated because of the way the analog to digital conversion takes place. As described in Section 5.1.1, we set the $GO/\overline{DONE}$ bit to start the conversion and then, we check thab bit until it gets cleared by the microcontroller. The code for this specific operation looks like this:

```
        // start conversion
        ADCON0bits.GO_DONE = 1;
        // loop until the conversion is finished
        while (ADCON0bits.GO_DONE != 0)
            Nop(); // how many time do we spend in here?
```

As we saw in that code snippet, there is no way to know *a priori* how many iterations are performed beforehand. Considering that, for each main loop iteration, there are three A/D conversions, is fundamental to find a method to determine exactly the number of iterations made in each busy-wait loop, otherwise the overall accuracy of the Three-Loop algorithm will progressively worsen.
The proposed solution is all about creating at the beginning of the main loop, one variable for each loop and substitute the **Nop()** calls with one of those variables, which gets incremented at each busy-wait cycle:

```
        int adc_loop_counter = 0;
        ...
        ...
        // start conversion
        ADCON0bits.GO_DONE = 1;
        // loop until the conversion is finished
        while (ADCON0bits.GO_DONE != 0)
            acLoopCounter++; // count the iterations made
```

So now it is only matter of opening the listing file created during the compile process, and count the number of instructions needed to implement the *busy-wait* loop. For example, the portion of disassembly listing referring to a_c acquisition looks like

```
                  // loop until the conversion is finished
                  while (ADCON0bits.GO_DONE != 0)
000114   a2c2   BTFSS   0xc2,0x1,0x0
000116   d006   BRA     0x124
000122   d7f8   BRA     0x114
                     acLoopCounter++;
000118   0e1e   MOVLW   0x1e
00011a   2adb   INCF    0xdb,0x1,0x0
00011c   0e1f   MOVLW   0x1f
00011e   e301   BNC     0x122
```

```
000120    2adb    INCF     0xdb,0x1,0x0
000124    50c4    MOVF     0xc4,0x0,0x0
```

$$T_{adc_loop} = [adc_loop_counter \cdot 9 + 1] \cdot T_{CY} \tag{5.8}$$

There is an other problem however: when we opened the listing file to count the number of instructions needed to implement the *busy-wait* cycle, we noticed the listing is composed of almost a thousand lines of code. That's because the *PIC* doesn't support natively floating point operations, so every time we did a convertion to a `float` type, or made an arithmetical operation where one of the operandw was a `float`, we actually called a particular routine provided by Microchip®, which emulated in software the floating point operation. So now we have also to take into consideration the delay produced by those algorithms, count them, multiply by their execution time listed in [2] and sum everything.
After cleaning up the listing, counting the floating point routines, the standard instructions, and considering the three delays to acquire the analog input, the fixed delay equals to $187.3\mu s$.

The complete algorithm is shown in Appendix A

5.2 *μLab* environment setup

Now we are finally able to build up our test environment: we added one embedded platform, which repesents the microcontroller being used, then we added the main matlab script responsible to update n_B and $\Delta\dot{\theta}$, based on $\Delta\delta$ input given by the controller, called `missile_aeroframe_2D.m`. We also added other two utility scripts: `duty2rad.m`, which converts the duty-cycle value to radians, and `normalizer.m`, which normalizes the model output to the range suitable for being passed back to the controller.
Those components communicates to each other by means of the so called *links*, used to connect the variables and parameters specified in each component.

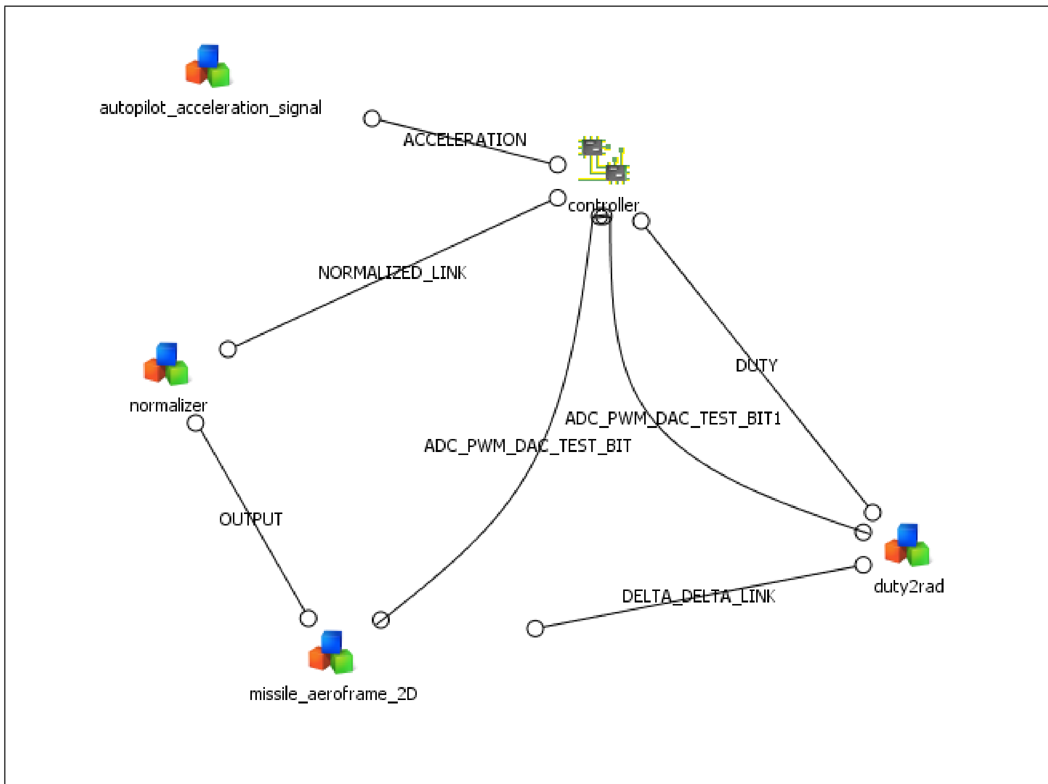
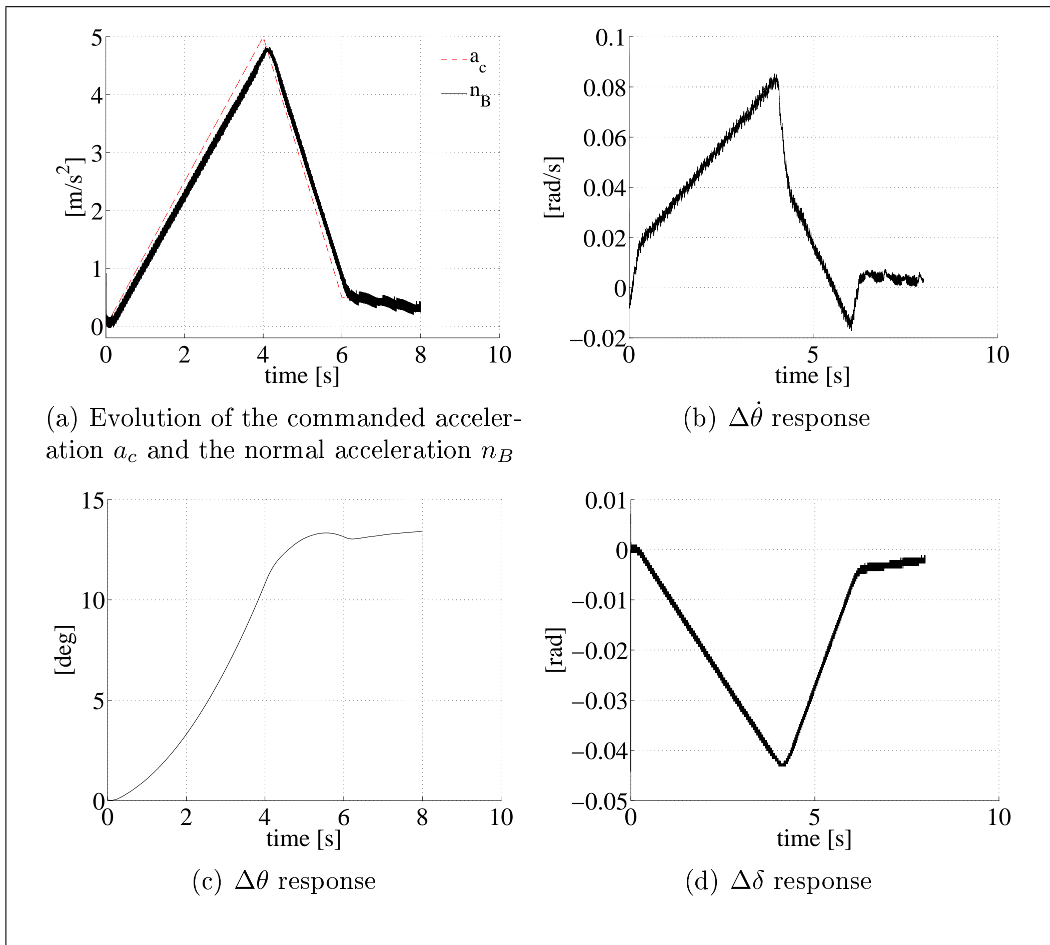


Figure 36: Testbench overview.

We could have inserted the two supplementary scripts into the main one, but this representation fits better the real case where the signals usually need to be adjusted by external components such as amplifiers or attenuators before being ready to be used. The complete script is shown in Appendix B

5.3 Co-Simulation results

In this section we finally are able to test the interactions between the firmware written and the matlab model of the missile, in absence of noise. We stopped our simulations after 8 seconds because it tooked about two hours to simulate that time span, so we opted for stopping the simulations at half of the last decreasing ramp. As we can see, the delay between the acceleration n_B



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

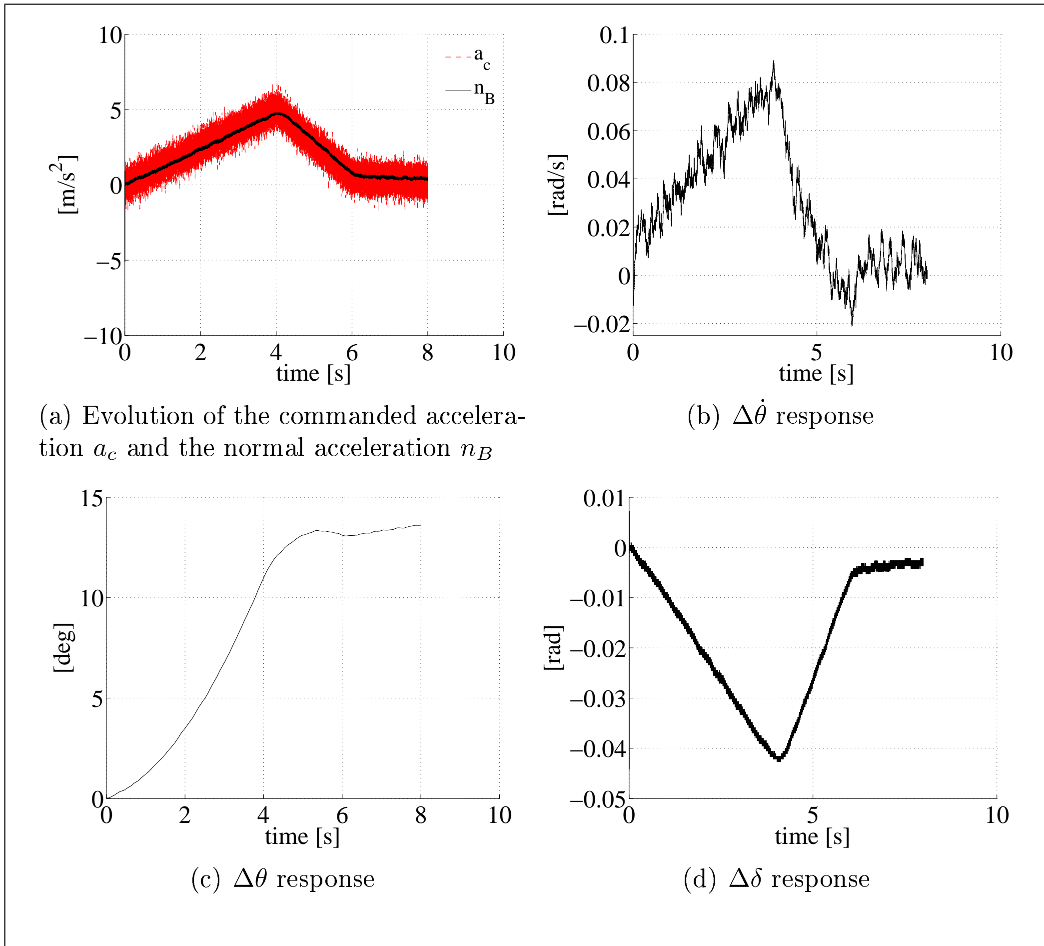
Figure 37: Overall evolution of the signals characterizing the algorithm.

and the commanded acceleration a_c is even more noticeable. Due to the quantizations taking places in the *PIC* device, n_B response is even more noisy compared with what we would expect from the matlab only simulation, even though it is still acceptable for our goal.

5.4 Noise disturbance effects

Similarly to what we did in Chapter 4, we want to test the performances of the compound system when a source of AWGN noise interacts with one of the connections between missile and controller. We will repeat the simulations first with noise disturbance $\sim N(0, 0.25)$ and then with noise disturbance $\sim N(0, 0.0025)$.

5.4.1 Noise disturbance $\sim N(0, 0.25)$ applied to a_c



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

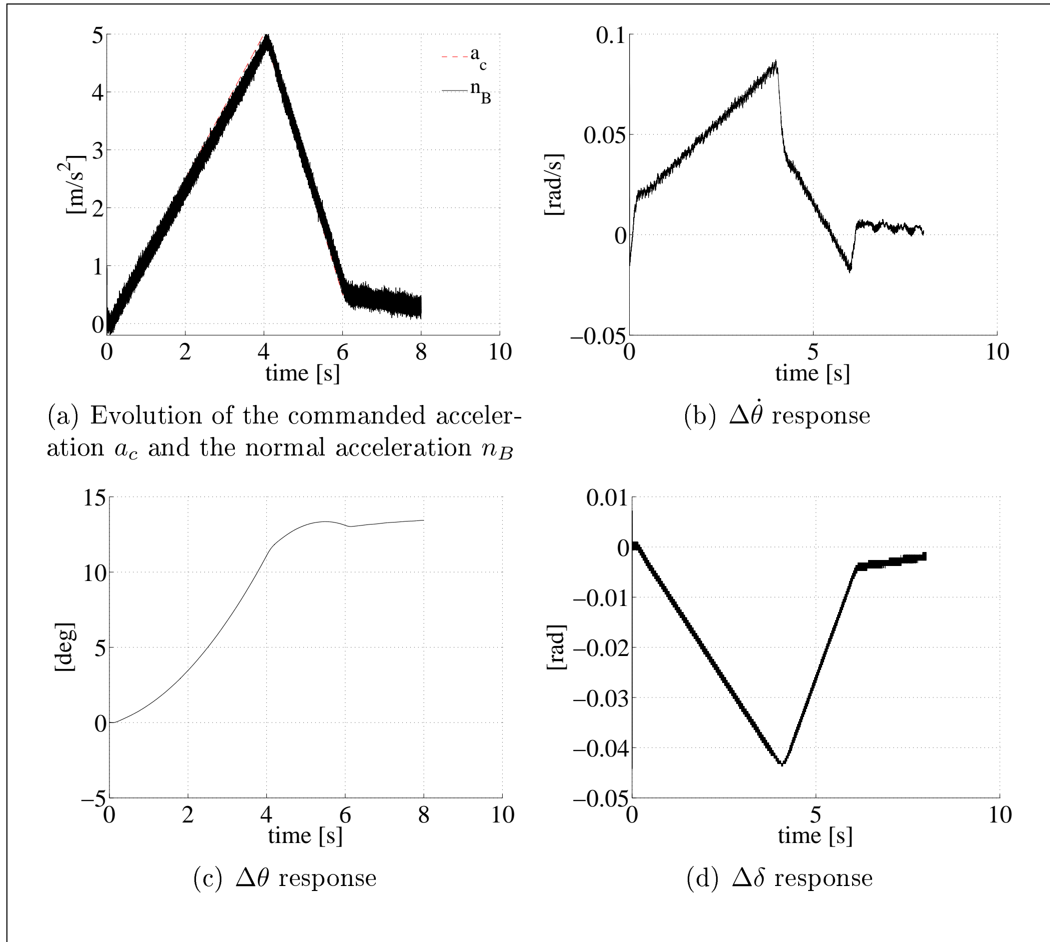
(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 38: Overall evolution of the signals characterizing the algorithm, with noise applied to a_c.

Even though a_c signal is very noisy, the feedback signal obtained $\Delta\delta$ looks slightly noisy, producing a smooth n_B response.

5.4.2 Noise disturbance $\sim N(0, 0.25)$ applied to n_B



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 39: Overall evolution of the signals characterizing the algorithm, with noise applied to n_B.

We can notice how those evolutions almost resembles the ones described in section 4.4.2; a slightly noisy ripple is present here, which is more noticeable in the $\Delta\delta$ response.

5.4.3 Noise disturbance $\sim N(0, 0.25)$ applied to $\Delta\delta$

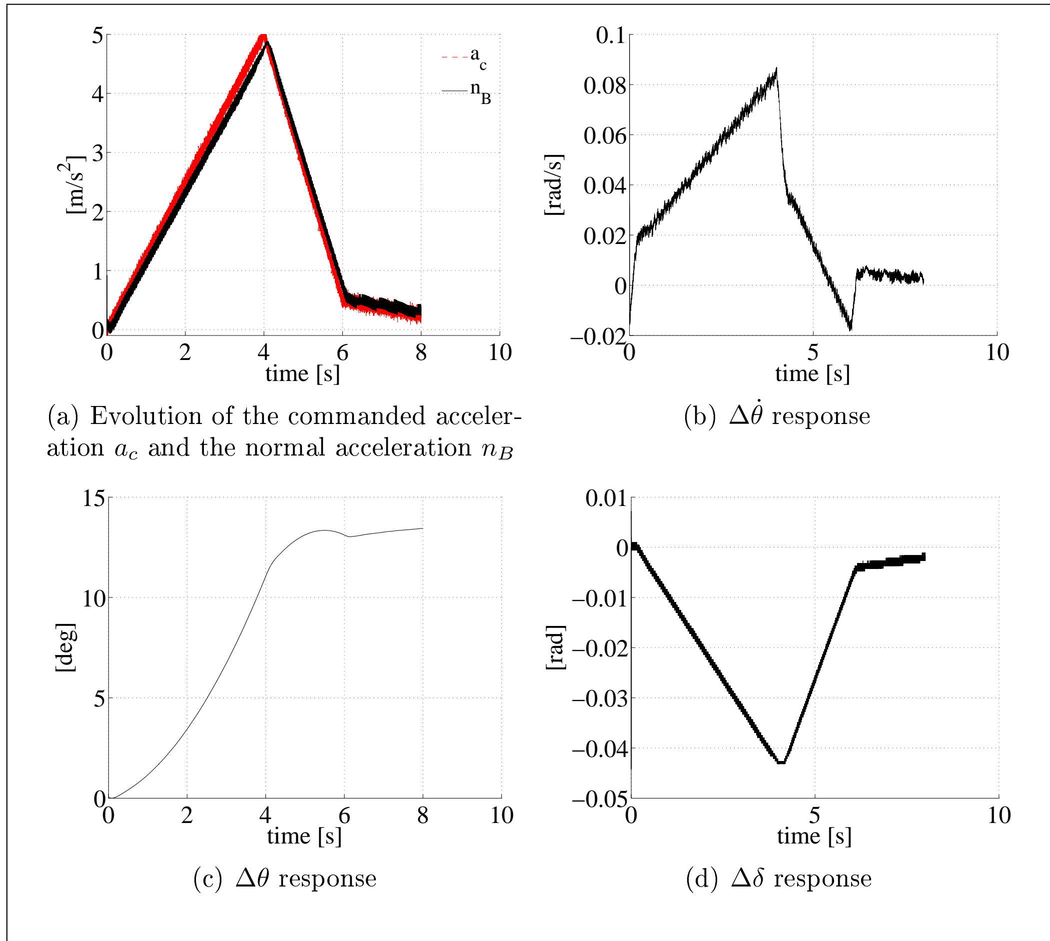
In Chapter 4 we saw how this particular case was the worse ever run, because the nature of the signal $\Delta\delta$ makes it higly sensitive to external noise sources. Anyway, moving to the co-simulation approach forced us to consider every detail about how the microcontroller works, and how it interacts with the surrounding environment. In this simulation for example, simulating a source

of noise applied to $\Delta\delta$ is completely useless, because $\Delta\delta$ is not affected at all by such noise: here is why.
As explained in Section 5.1.3, PIC devices doesn't ship with digital to analog output converter so we had to use the PWM module to perform a conversion trick: internally, we converted $\Delta\delta$ into an integer value which is used to modulate a pwm waveform. Externally, this waveform gets captured by a supplementary device which perform a conversion from the duty-cycle to the original value, which is sent to the missile model. So now the information is no longer binded to a voltage value as it happens for the inputs; instead, the information is binded to how much time the waveform stays in its *high* state, or *logic "1" level.* This difference in how the information gets transmitted is crucial because of how the levels in a TTL/CMOS device are treated:

$$\begin{cases} \text{logic level 1} & \text{if } voltage \in [3.3, 5] \text{ Volts} \\ \text{logic level 0} & \text{if } voltage \in [0, 0.8] \text{ Volts} \end{cases} \quad (5.9)$$

So it's easy to see that noises with amplitude $\pm 10\%$ of the microcontroller operational range, that is $[0, 5V]$, are not sufficient to trigger a wrong logical level detection. Of course this won't be always true in a real world application because logic ports suffers of a certain amount of bias but still, we are referring to an ideal situation where a 0 logic level means $0Volts$ and a logic level 1 refers to $5Volts$; in any case, such a big source of noise is highly unlikely to happen so these assumptions are most likely true.

5.4.4 Noise disturbance $\sim N(0, 0.0025)$ applied to a_c



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

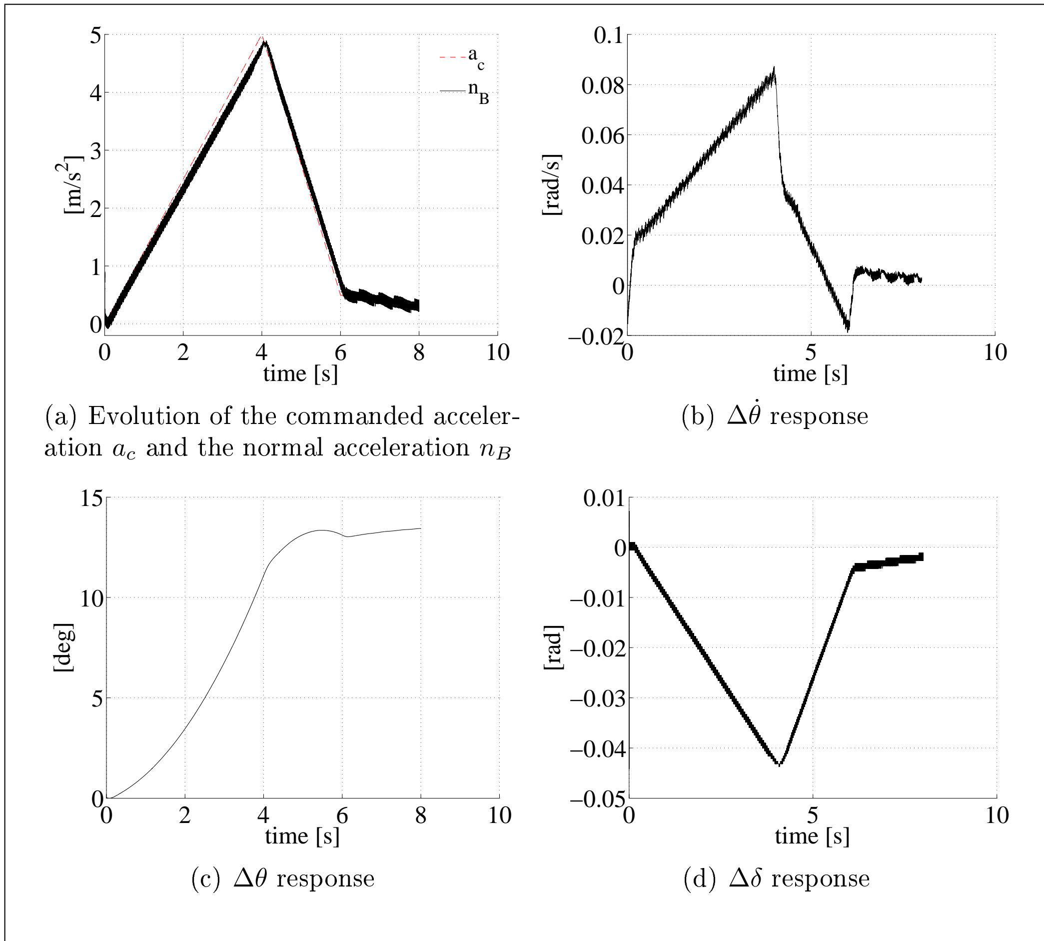
(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 40: Overall evolution of the signals characterizing the algorithm, with noise applied to a_c.

Here, again, the responses are almost equals to the pure matlab simulation counterpart, with a more evident delay present here due to the algorithm execution.

5.4.5 Noise disturbance $\sim N(0, 0.0025)$ applied to n_B



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

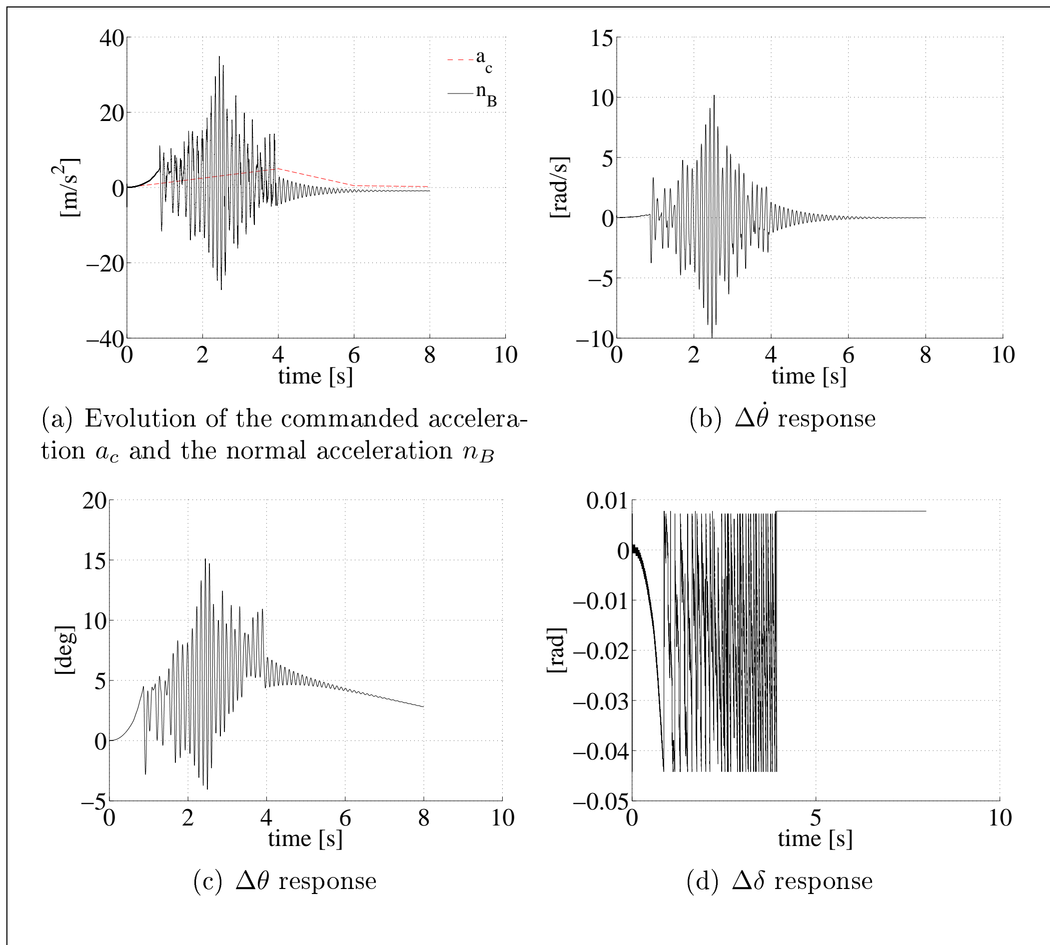
Figure 41: Overall evolution of the signals characterizing the algorithm, with noise applied to n_B.

The responses here start being different compared to the simulation made in a pure matlab environment: n_B evolution is more noisy, the same observation holds for the other signals too.

5.5 Faulty conditions analysis

We will repeat now the analysis of the most common faulty condition that could happen since we are interested about how the microcontroller will react to such situations.z

5.5.1 Absence of n_B



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

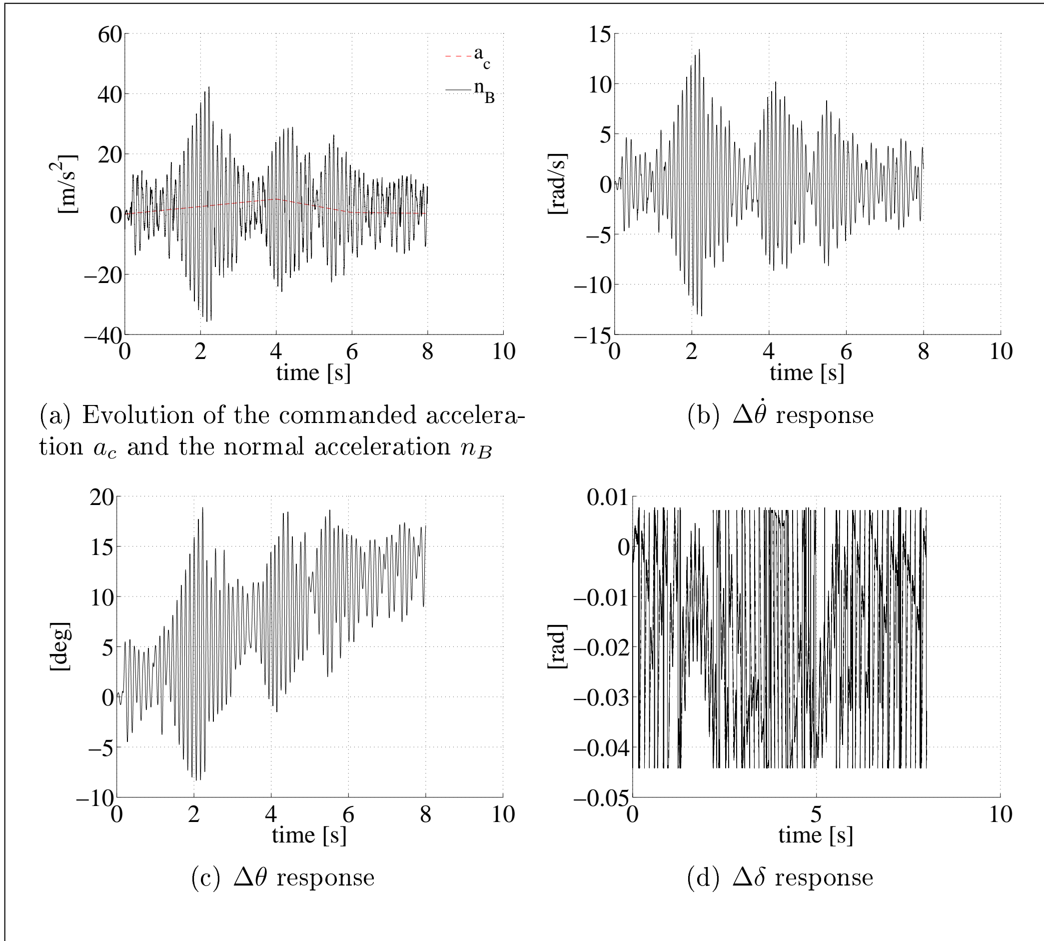
(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 42: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\dot{\theta}$.

We can see as, with no signal n_B available, the controller produces completely erratic responses.

5.5.2 Absence of $\Delta\dot{\theta}$



(a) Evolution of the commanded acceleration a_c and the normal acceleration n_B

(b) $\Delta\dot{\theta}$ response

(c) $\Delta\theta$ response

(d) $\Delta\delta$ response

Figure 43: Overall evolution of the signals characterizing the algorithm, with noise applied to $\Delta\dot{\theta}$.

Here the response doesn't look like what we got in the matlab simulation counterpart, but this shouldn't be that surprising because the microcontroller limits the inputs it receives, thus preventing them to diverge indefinitely.

6 Conclusions and further development

Even though the two different simulation approaches showed similar results in most of the tests performed, the co-simulation one provided a better understanding of the underlying hardware in use. Without this approach we would never notice the problems which necessarily arises when we mix physical objects governed by electronics devices until the moment when the firmware gets tested for the first time on a tangible system. Of course this thesis is not and ending point, but a good point to start with: there's a lot room for improvements like testing better types of controllers like the classic PID, or adopting a space-state controller, or even trying to implement a Model Predictive Control (MPC). And the best way to get accurate and reliable results, as the complexity increases, is with no doubt the adoption of the Co-Simulation approach.

7 Acknowledgements

As this chapter of my life turned to its end, it was inevitable to think about everything that happened to me in these years. Lots of joyful and, sometimes, sorrowful moments; but I've always been surrounded by awesome people that cheered me up when I was letting me down, and shared my happiness when I succeeded in something important for me.
So, it's time to say a huge and passionate *"Thank You"* to all these great guys/gals :)
The first *"Thank You"* of course goes to my parents: without their support and their confidence in me, I wouldn't never reach such important destination all alone.
The same goes for my brother Matteo and his fiancée Evi, randomly popping out in my room while I was about to set my computer on fire because of these evilish simulations, and handing me some praline or chocolate candy to cheer me up.
Of course I have to thank my grandparents, uncles and aunties, cousins; they were always interested in my progress and eager to come here today and celebrate this important goal achieved.
I have to say *"Thank You"*, and also *"I'm sorry"*, to my beloved fiancée Mariaelena, because of her endless patience and support. So many times I've been detached and annoyed, especially in these last chaotic months, because of people I had to deal with whom pissed me off. I have to thank her for withstanding the nerd/geek side of me, for the joyful moments spent together and for the others that will come.
An other huge *"Thank You"* goes also to Alberto and his brother Francesco: they are irreplaceable friends, they supported me as much as my fiancée did and does. I will never forget the craziest and funniest drinking spree I've ever had: the day after I was up at 8.30 a.m., studying Signal and Systems, without any tangible sign of hangover. We really needed to move on and leave our concerns behind us, and that night certainly we made it :) Also, thank you Alberto for introducing me to the Linux world: it changed my life, really. Anyway, come back to KDE!
"Thank You" Riccardo too, we had a lot of funny and embarassing moments together, like the reinterpretation of the *bubble*-sort algorithm, or the famous quote " With great *power* comes great responsibility", you know what I'm talking about ;) It's a shame that you never, ever, listened to my advices on how to start a proper chat with a girl.. Such precious advices thrown in the wind! Jokes aside, it's always a pleasure to talk with you and I'm glad for the moments we spent.
Thank you Nicola for the nerdy jokes made during the classes we shared,

and the knowledge about everything we wondered about, and for all my programming-related doubts you helped me to understand.
Thank you Simone for hosting the most awesome lan-parties I've ever joined, and of course I have to thank again Alberto, Francesco, Nicola, Simone, and also Ivan, Daniele, Edoardo, Alessandro, Andrea (Bordy) and Andrea (Larry) for sharing the fun all together. Also thank you Simone for the copy of Visual C++ 2003, *100% genuine*, you gave me: that's how my programming life started, you fool! :)
Thank you Ivan and Alessandro for the hours spent playing Quake III Arena, and related chitchat during Ciscato's class on the perfect strafe-jump. I'm a pro now!
Thank you Daniele and Edoardo for the time spent talking about the awesome character personified by Mr. Jonh Glover, the fun with Mosconi's speeches, and the hours of serious study we spent together.
Manuel, thank you for the explanations you gave me back in high school about Dragon Ball, and the mangas read back at that time. It was a hard time for us, but we also had funny moments playing PGA Tour or chess during Emilio's lessons.
Thank you Stefania for being a true friend, and for all the coffee breaks at the bar/ general hospital we made. I was really sorry for the idiots that hurted your feelings in the past; thankfully, now you are happier than ever. I'll never forget our crazy holiday in Spain: Girona, Barcelona, Zaragoza, Madrid and Valencia, five cities in one week.. I Dunno how we managed to survive, anyway we did it!
Also, thank you Alessandra (and the others above) for the funny moments we had at Jappelli, the chat about fashion and drinks, and the "you are gloomy and lonely people" poster.
Giulia, thanks for for trying to make Mariaelena attend our classes, and agreeing that Pierobon is one of the most awesome professor ever had even if, at first, we didn't understand that much of what he was talkin about.

From the working side of my life, I have to say a huge *"Thank You"* at Walter and Sergio for their unconditional help and support.
I've got to thank also Lino, Roberto, Mirco, Fabio, Sergio, Victor and Franco, my ex co-workers, and the craziest crew you will ever meet, I mean it :) The work was tough but at the end of the day you were always going home with a smile on your face, remembering some amusing episode happened.

Last but not least at all, I have to say *"Thank You"* to Google for sponsoring my work as programmer for two consecutive summers, and of course KDE because choose me as student eligible to get Google's sponsorhip, and

for providing the best Linux Desktop Environment out there. It's been a pleasure to work with you, and since my academic life is over now, I hope to contribute back again.

I want also to say *"Thank You"* to all the people that never trusted in me, in what I was about to do, and in the choices I made. There's nothing more rewarding than succeed despite of other's expectations.

A Complete Three-Loop Algorithm

```
/**
  * Author: Diego Casella
  */

/// Includes
#include <p18f4620.h>
#include <delays.h>                 // Needed for Delay10KTCYx

/// Configuration bits
#pragma config OSC = XT             // Use external oscillator
#pragma config WDT = OFF            // WatchDog Timer turned off
#pragma config PWRT = ON            // Power-up timer turned on
#pragma config MCLRE = OFF
#pragma config LVP = OFF

/*
 * Macro which tests if theValue is in the [min, max] range.
 * If not, sets the appropriate LATD port accordingly
 */
#ifdef __DEBUG
    #define TEST_BOUNDS(theValue, min, max, minPort, maxPort)\
        LATDbits.LATD##minPort = 0;     \
        LATDbits.LATD##maxPort = 0;     \
        if(theValue < min) {            \
            LATDbits.LATD##minPort = 1; \
        }                               \
        if(theValue > max) {            \
            LATDbits.LATD##maxPort = 1; \
        }
#else
    #define TEST_BOUNDS(theValue, min, max, minPort, maxPort) \
                        ;
#endif


/*
 * main program here
 */
void main(void) {
```

```
/// Setup vars, ADC and PWM modules
unsigned long int adcResult;
double adcConvertedResult;
double yEulerPrevious = 0;
// 28 fp instructions + 176 1-cycle instructions + 3 delays
double time = 0.0001873;
double result = 0;
unsigned int duty = 0;
double tcy = 0.0000001;
double _time = 0;
int acLoopCounter = 0;
int nbLoopCounter = 0;
int thetaLoopCounter = 0;


/// setup ADC
// set RA0..2 as inputs
TRISA = 0b00000111;
// 8*Tosc and 12*Tad, left-justified
ADCON2 = 0b00101001;
// Vref- = Vss, Vref+ = Vdd, and set
// AN0..2 as analog pins
ADCON1 = 0b00001100;
// select AN0, turn on ADC and put it
// enabled and idle state
ADCON0 = 0b00000001;

// debug purpose only
TRISD = 0xff;

// Set RC2 initial state
LATCbits.LATC2 = 0;
/// ADC test enabled?
LATAbits.LATA3 = 0;


/// setup PWM
CCP1CON = 0b00001100;
// We want RC2 acting as PWM output
TRISC = 0b11111011;
// prescaler = 1, activate TMR2
```

```
T2CON = 0b00000100;
// period = 255
PR2 = 0b111111111;
// set duty cycle = CCPR1L / PR2
// note: period = (PR2+1) * Tcy * prescaler
CCPR1L = 0b00000000;

/// disable adc/pwm/dac test output port
LATAbits.LATA3 = 0;

/// Starting Three-Loop algorithm
while (1) {
    // reset the counters first
    acLoopCounter = 0;
    nbLoopCounter = 0;
    thetaLoopCounter = 0;
    _time = 0;
    /// reading the autopilot acceleration
    ADCON0bits.CHS0 = 0;
    ADCON0bits.CHS1 = 0;
    // wait about 20us to charge AD capacitor
    // NOTE: too much delay, move in a ISR!
    Delay10TCYx(20);
    // start conversion
    ADCON0bits.GO_DONE = 1;
    // loop until the conversion is finished
    while (ADCON0bits.GO_DONE != 0)
        acLoopCounter++;


    // now read from ADRESH e ADRESL
    adcResult = ((unsigned int)ADRESH << 2) |
                 ((unsigned int)ADRESL >> 6);

    // convert the result back into [0, 5] range
    // note: 0.00488 = 5/1023
    adcConvertedResult = adcResult * 0.004887585;

    // computing result_1
    // g2acc1 * KDC = 9.81*1.0531
    result = adcConvertedResult * 10.33075;
```

```
        // Test result_1
        TEST_BOUNDS(result, 0, 52, 0, 1)


        /// reading nB from AN2
        ADCON0bits.CHS1 = 1;
        // wait about 20us to charge AD capacitor
        Delay10TCYx(20);
        // start conversion
        ADCON0bits.GO_DONE = 1;
        // loop until the conversion is finished
        while (ADCON0bits.GO_DONE != 0)
            nbLoopCounter++;


        // now read from ADRESH e ADRESL
        adcResult = ((unsigned int)ADRESH << 2) |
                    ((unsigned int)ADRESL >> 6);

        // convert the result into degreees
        // note: 0.04887585 = 50/1023
        adcConvertedResult = adcResult * 0.04887585;
        // Test result_2
        TEST_BOUNDS(adcConvertedResult, -0.1, 50, 2, 3)


        /// 1st loop
        result -= adcConvertedResult;

        // multiply by KA
        result *= 0.02743794;


        /// reading deltaTheta from AN1
        ADCON0bits.CHS0 = 1;
        ADCON0bits.CHS1 = 0;
        // wait about 20us to charge AD capacitor
        Delay10TCYx(20);
        // start conversion
        ADCON0bits.GO_DONE = 1;
        // loop until the conversion is finished
```

```
while (ADCON0bits.GO_DONE != 0)
    thetaLoopCounter++;


// now read from ADRESH e ADRESL
adcResult = ((unsigned int)ADRESH << 2) |
             ((unsigned int)ADRESL >> 6);

// convert the result into degreees
// note: 0.000127 = (5/1023)*0.026
adcConvertedResult = adcResult * 0.00012707 - 0.03;
// Test result_3
TEST_BOUNDS(adcConvertedResult, -0.02, 0.1, 4, 5)

/// 2nd loop
result = adcConvertedResult - result;

// multiply by WI
result *= 32.34312;

// euler integration
// y(k) = y(k-1) + u(k)*time
// time spent waiting ac conversion
_time = (acLoopCounter*9+1)*tcy;
// + time spent waiting nB conversion
_time += (nbLoopCounter*9+1)*tcy;
// + time spent waiting theta conversion
_time += (thetaLoopCounter*9+1)*tcy;
result = yEulerPrevious + result * (time + _time);
yEulerPrevious = result;

/// 3rd loop
result += adcConvertedResult;
// dont perform result*KR*10, signal too
// small and noisy
// Test result_3
TEST_BOUNDS(result, -0.6, 0.1, 6, 7)

// convert before send to PWM
result += 0.6;
result *= 1451.42;
```

```
        // convert double to int
        duty = (int) result;

        // set duty cycle
        // note: duty = CCPR1L * Tosc * prescaler
        CCP1CONbits.DC1B0 = 0b00000001&duty;
        CCP1CONbits.DC1B1 = 0b00000001&(duty>>1);
        CCPR1L = duty>>2;
    }
}
```

B Complete missile_aeroframe_2d script

```
function [component] = matlabModel(component, context)
  %% Reading muLab parameters and variables
  globalTime = context.globalTime
  % Delta fin angle
  DELTA_DELTA = getfield(component.variables,'DELTA_DELTA')
  ADC_PWM_DAC_TEST_ENABLED = ...
    getfield(component.variables,'ADC_PWM_DAC_TEST_ENABLED')
  if ADC_PWM_DAC_TEST_ENABLED == 5
      % we're just interested in testing the ADC module
      return;
  end

  %% Pre-computed parameters (improves speed)
  %% We define all these variables as 'global' at the very
  %% first simulation step, so they are cached globally
  %% instead of begin re-created/computed in every function
  %% invocation
  global index;
  global time;
  global delta_delta;
  global delta_delta_s;
  global TFnd;
  global TFqd;
  global integrator;

  if globalTime == 0
```

```
    index = 1;
    DENnd = [307.888604797530e-006 ...
             745.670831812253e-006 ...
             1.00000000000000e+000];
    DENqd = [307.888604797530e-006 ...
             745.670831812253e-006 ...
             1.00000000000000e+000];
    NUMnd = [349.637890026630e-003 ...
             0.00000000000000e-003 ...
             -1.11796938975997e+003];
    NUMqd = [-1.35528510971017e+000 ...
             -1.62833381004383e+000];
    TFnd = tf(NUMnd,DENnd)
    TFqd = tf(NUMqd,DENqd)
    integrator = tf([1], [1 0]);

    % lsim requires at least two samples, fake it just for now
    time = [0 context.samplingPeriod];

    % Same with the delta_delta vector, but we just keep the
    % second value equal to zero.
    delta_delta = [0 0];

elseif index == 1
    % Update just the index, don't increase the array because
    % it's already of size 2
    index = index + 1;
    time(index) = globalTime;
    delta_delta(index) = DELTA_DELTA;
else
    % Update the index, then grab globalTime and DELTA_DELTA,
    % save them in the vector of the times and inputs so lsim
    % can advance its simulation.
    % Note: growing arrays is a DAMN SLOW operation!
    index = index + 1;
    time = [time globalTime];
    delta_delta = [delta_delta DELTA_DELTA];
end

% simulate the response of TFnd and TFqd
ub = lsim(TFnd, delta_delta, time);
```

```
  delta_theta = lsim(TFqd, delta_delta, time);

  % Retrieve results
  DELTA_THETA = delta_theta(index)
  NB = ub(index)

  % save results
  component.variables = setfield(component.variables,'NB', NB)
  component.variables = setfield(component.variables,
                                'DELTA_THETA', DELTA_THETA)
end
```

References

[1] *PIC18F2525/2620/4525/4620 Data Sheet*, ch. 19.0 10-Bit Analog-To-Digital Converter (A/D) Module, Microchip, 2004.

[2] *PIC18F2525/2620/4525/4620 Data Sheet*, ch. 26.4.3, Table 26-25: A/D Conversion Requirements, Microchip, 2004.

[3] *PIC18F2525/2620/4525/4620 Data Sheet*, ch. 16.4, Enhanced PWM Mode, Microchip, 2004.

[4] D. Ciscato, *Appunti di Controllo Digitale*, Libreria Progetto, Via Gradenigo 2, 2010.

[5] Benvenuto N., Corvaja R., Erseghe E., and Laurenti N., *Communications Systems - Fundamentals and Design Methods*, Wiley, 2007.

[6] Da Forno R, *01 - Dinamica 2D Missle*, unpublished, slides.

[7] Da Forno R., *03 - Strategie di Controllo Missile 2D*, unpublished, slides.