Altitude control of the nano quadcopter Crazyflie

Émilie THOMÉ
emilie.thome@student.isae-supaero.fr
Institut Polytechnique de Paris
Palaiseau, FRANCE

ABSTRACT

As critical functions increased in the implementation of unmanned aerial vehicles (UAV) system, a formal verification of their control system is necessary to assure safety.

Traditionally, tests and simulations are run on embedded systems to ensure correctness. However, there is no concrete proof that demonstrate safety, only statistics. While formal verification ensures rigorous correctness of the code by using formal methods techniques. Those formal methods represent a promising solution for code verification. And as the amount of code requiring proofs increases, the automation of verification processes through formal methods is likely to develop.

This report is an introduction to formal verification of UAV control system. The subject of this study is the nano quadcopter Crazyflie and its altitude controller. This nano drone is an open-source project developed by Bitcraze. Its firmware is written in C++ and and is available on GitHub.

The aim of this project is to formally verify the Crazyflie's altitude controller by modelling it and demonstrating its stability.

CCS CONCEPTS

• **Computer systems organization** → **Robotic control**; • **Hardware** → *Safety critical systems.*

KEYWORDS

UVA, quadcopter, controller, stability, formal methods

ACM Reference Format:
Émilie THOMÉ. 2021. Altitude control of the nano quadcopter Crazyflie. *ACM Trans. Graph.* 0, 0, Article 0 (February 2021), 11 pages. https://doi.org/0

1 INTRODUCTION

1.1 The Crazyflie

First of all, an introduction to the Crazyflie is needed.

The Crazyflie is a nano (92x92x29mm) quadcopter developed by Bitcraze. As it was created as a development platform, its code and design specification were made accessible[BitCraze 2020]. It has a flight time of 7 minutes and a charge time of 40. It can be extend with additional boards that are connected through two rows of pin headers.

Author's address: Émilie THOMÉ, emilie.thome@student.isae-supaero.fr, Institut Polytechnique de Paris, Palaiseau, FRANCE, 91764.

Figure 1: Crazyflie

All of those characteristics and its affordability make it a very useful tool in the research field. And therefore it is very well documented, documentation that I used during this project.

1.2 Formal verification

Formal verification is proving the correctness of an algorithm directing a system's evolution, with certain specifications or properties, using formal methods. Formal methods are mathematically based techniques that aim to specify, develop or validate programs. They are applications of theoretical computer science fundamentals which are mathematical topics of computing. They mainly use formal languages like mathematical logic or theorem to specify properties in systems and programs. A specific formal method is defined by the theoretical background it uses to prove a program.

In this paper is presented the method used to prove the correctness of the Crazyflie's altitude controller. First, a study over the controller's stability is proceed will give conditions to satisfy in order to achieve a certain property. Following, a reachability analysis will make it possible to focus the case study to selected situations and then meet the previous conditions.

1.3 Objectives

A drone is supposed to maintain its position very easily. In this paper, we focus on how the Crazyflie maintains its altitude (no roll, no pitch, no yaw). To do so, the drone needs a control system.

So the main objectives of this project are to find how the Crazyflie controls its altitude and to show the stability of its altitude controller.

For that, it will be necessary to :

- define a way to check the stability of a system,
- model this controller,
- demonstrate controller stability altitude of the UAV.

2 FORMAL METHODS

In this section are described the formal methods used to elaborate the verification of the Crazyflie's altitude controller.

2.1 Stability

Different kind of stability can be used to describe solutions of differential equations describing dynamical systems. In order to measure the ability of a system to stay at a given altitude, the stability of solutions near to a point of equilibrium is the natural stability. It is the Lyapunov stability.

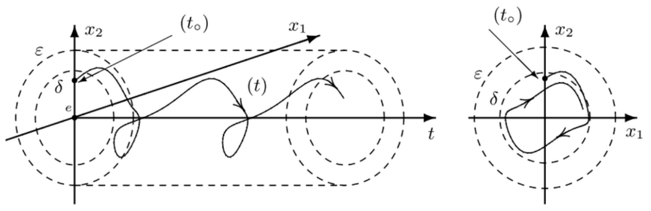


Figure 2: 2015 Hassani Walid. Notion of Lyapunov stability

A system is Lyapunov stable for an equilibrium point x_e if all movement in the neighborhood of x_e remains in the neighborhood of this point.

The Lyapunov stability is the stability used in this study to verify the correctness of the Crazyflie's altitude controller. The following theorem was used to prove the stability of our system.

Lyapunov's stability theorem: Let a system described by a differential equation of the type $\dot{x} = f(x,t)$. Without loss of generality, the equilibrium point of interest is $x_e = 0$. If there is a function $V : (x,t) \longrightarrow V(x,t)$ continuously differentiable defined positive such that $\dot{V}$ is semi defined negative and that $V(x,t) \xrightarrow[||x||\to\infty]{} \infty$ then 0 is a stable equilibrium point. If V satisfies the previous conditions, then V is a Lyapunov function.

These conditions are similar to those that potential energy must verify for an equilibrium point of physical system to be stable.

2.2 Reachability analysis

Following the stability study, we will encounter some difficulties finding a function that satisfies all the conditions to be a Lyapunov function for the entire state space. We will bound our state study space, using the reachability analyzer for non-linear (hybrid) systems Flow* [Xin Chen 2017].

Flow* makes it possible to compute an over-approximation of the reachable states of an hybrid automaton from an initial state. Given a bounded time interval Δ and a natural number n , Flow* computes a finite number of Taylor models containing all states which are reachable in the time Δ (via at most m jumps for hybrid systems). If a given state is not included then it is absolutely unreachable.

3 MODEL

3.1 Modelling the Crazyflie altitude controller

We are interested in altitude control with the following variables:

- the altitude of the center of gravity in the inertial frame: z,
- the vertical velocity of the center of gravity in the inertial frame (no rotation, only the altitude change): v_z.

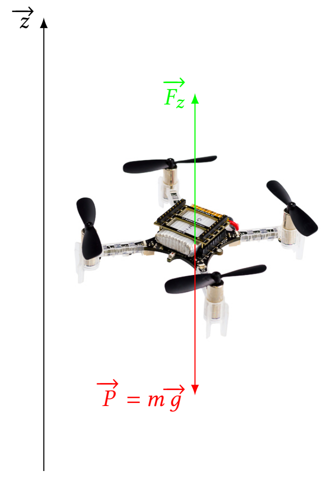
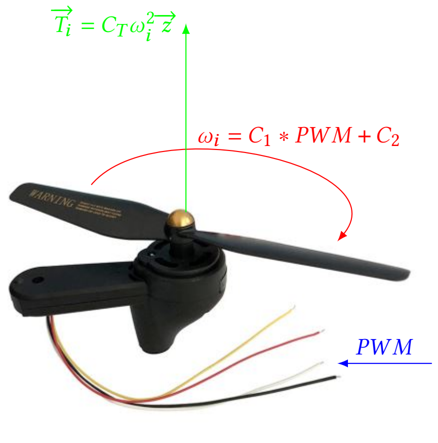


Figure 3: Simple Crazyflie physical model

According to the fundamental principle of dynamics:

$$\sum F = m\dot{v_z} \text{ then } \dot{v_z} = \frac{F_z}{m} - g$$

The force generated by each propeller according to z is of the form $T_i = C_T\omega_i^2$, ω_i being the rotation speed of the i^{th} motor (in revolutions per minute) and C_T the thrust coefficient. By symmetry we can induce the equality of rotation speeds $\omega_1 = \omega_2 = \omega_3 = \omega_4 = \omega$ and $F_z = 4 * C_T\omega^2$.



$\vec{T_i} = C_T\omega_i^2\vec{z}$
$\omega_i = C_1 * PWM + C_2$
PWM

Figure 4: Propeller

The DC voltage sent to each motor is controlled using a 16-bit PWM signal (from 0 to 65535). The real input is therefore this PWM signal. The relationship between PWM and the angular speed of the motors ω (in revolutions per minute) is apparently linear (according to experiments):

$$\omega = C_1 * PWM + C_2$$

Then, we have :

$$F_z = (4C_T C_1^2) * PWM^2 + (8C_T C_1 C_2) * PWM + 4C_T C_2^2$$

However, we will use the values found experimentally in [Förster 2015] :

$$F_z = 4(k_1 * PWM^2 + k_2 * PWM + k_3)$$

Let an instruction z_c, the quadricopter is controlled by two Proportional Integral controllers (according to the firmware codes in appendix) :

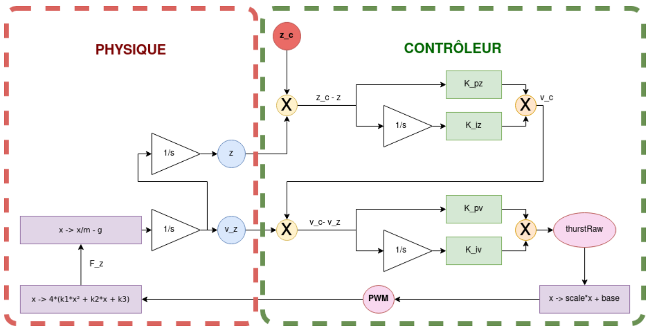


Figure 5: Altitude controller diagram

The transition from the physical part to the controller part has not been dealt with in this paper. It is done by a Kalman estimator (or a complementary one, from what I saw in the firmware code).

If we note $K_{pz} = 2$, $K_{iz} = 0.5$, $K_{pv} = 25$ and $K_{iv} = 15$ and also Scale = 1000 and Base = 36000, the equations we can extract from the firmware code are as follows:

- $v_c = K_{pz}(z_c - z) + K_{iz}\int(z_c - z)dt$
- $thrustRaw = K_{pv}(v_c - v_z) + K_{iv}\int(v_c - v_z)dt$
- $PWM = \text{Scale} * thrustRaw + \text{Base}$

Then, we have:

$$PWM = 1000 * \left[25\left(2(z_c - z) + 0.5\int(z_c - z)dt - v_z\right) + 15\int\left(2(z_c - z) + 0.5\int(z_c - z)dt - v_z\right)dt\right] + 36000$$

By adding two new states u_1 and u_2 representing respectively the integrals $\int(2(z_c-z)+0.5\int(z_c-z)dt-v_z)dt$ and $\int(z_c-z)dt$ and by introducing the new constants $K_p = \text{Scale}*K_{pv}$ and $K_i = \text{Scale}*K_{iv}$ we obtain something quite different (because it is not the same firmware) than [S. Putot 2019]:

$$\begin{cases} \dot{z} = v_z \\ \dot{v_z} = \frac{4k_1}{m} * PWM^2 + \frac{4k_2}{m} * PWM + \frac{4k_3}{m} - g \\ \dot{u_1} = 2(z_c - z) + 0.5u_2 - v_z \\ \dot{u_2} = z_c - z \\ PWM = K_p\dot{u_1} + K_i u_1 + \text{Base} \end{cases}$$

At equilibrium, $F_z = mg$ so $4(k_1 * PWM^2 + k_2 * PWM + k_3) = mg$. Numerically: $PWM_e \approx 37287 \approx \text{Base}$

The state equilibrium is reached when every derivative is zero.

$$\begin{cases} 0 = v_{ze} \\ 0 = 2(z_c - z_e) + 0.5u_2 - v_{ze} \\ 0 = z_c - z_e \\ PWM_e = K_i u_{1e} + \text{Base} \end{cases}$$

Then the equilibrium is:

$$\begin{cases} z_e = z_c \\ v_{ze} = 0 \\ u_{1e} = \frac{PWM_e - \text{Base}}{K_i} \approx 0.0858 \\ u_{2e} = 0 \end{cases}$$

3.2 Linearizing model equations around equilibrium

In order to linearize the equations, Taylor expansion is applied to the first order near equilibrium:

$$\begin{cases} \Delta\dot{z} = \Delta v_z \\ \Delta\dot{v_z} = \frac{8k_1}{m} * PWM_e * \Delta PWM + \frac{4k_2}{m} * \Delta PWM \\ \Delta\dot{u_1} = -2\Delta z + 0.5\Delta u_2 - \Delta v_z \\ \Delta\dot{u_2} = -\Delta z \\ \Delta PWM = K_p \Delta\dot{u_1} + K_i \Delta u_1 \end{cases}$$

Therefore, by handling the equations:

$$\begin{cases} \Delta\dot{z} = \Delta v_z \\ \Delta\dot{v_z} = (\frac{8k_1}{m} * PWM_e + \frac{4k_2}{m}) * (-K_p(2\Delta z - 0.5\Delta u_2 + \Delta v_z) + K_i\Delta u_1) \\ \Delta\dot{u_1} = -2\Delta z + 0.5\Delta u_2 - \Delta v_z \\ \Delta\dot{u_2} = -\Delta z \\ \Delta PWM = -K_p(2\Delta z - 0.5\Delta u_2 + \Delta v_z) + K_i\Delta u_1 \end{cases}$$

If we write x the vector $[z\ v_z\ u_1\ u_2]^t$, $\Delta x = [\Delta z\ \Delta v_z\ \Delta u_1\ \Delta u_2]^t$ and $\Delta\dot{x} = [\Delta\dot{z}\ \Delta\dot{v_z}\ \Delta\dot{u_1}\ \Delta\dot{u_2}]^t$ then we have:

$$\begin{cases} \Delta\dot{x} = A * \Delta x \\ \Delta PWM = B * \Delta x \end{cases}$$

with $\alpha = \frac{8k_1}{m} * PWM_e + \frac{4k_2}{m}$,

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -2K_p * \alpha & -K_p * \alpha & K_i * \alpha & 0.5K_p * \alpha \\ -2 & -1 & 0 & 0.5 \\ -1 & 0 & 0 & 0 \end{bmatrix} \text{ and}$$

$$B = \begin{bmatrix} -2K_p & -K_p & K_i & 0.5K_p \end{bmatrix}$$

3.3 Model constants

The physical constants recorded about the Crazyflie are in the table 1.

4 LYAPUNOV FUNCTION

4.1 A method to find a Lyapunov function

The method is to use the linearized system to solve the Lyapunov equation:

$$A^t P + PA = -Q$$

with A the matrix of our dynamics, P and Q symmetrical and defined positive. Q is chosen (at random), we deduce P. Being symmetrical, we can diagonalize P in order to verify that its eigenvalues are

m	0.028
k_1	$2.130295e-11$
k_2	$1.032633e-6$
k_3	$5.484560e-4$
Scale	1000
Base	36000
K_{pz}	2
K_{iz}	0.5
K_{pv}	25
K_{iv}	15
K_p	25000
K_i	15000

Table 1: Crazyflie model constants (in SI)

strictly positive. This ensures that the function $V : x \longrightarrow x^t P x$ is defined positive and that $-\dot{V} : x \longrightarrow -\nabla V A x = x^t Q x$ is too.

4.2 A Lyapunov function for the linearized system

There exists, in the python library `control`, a function `lyap` that solves the Lyapunov equation. I used it to compute the matrix P in the code fig.6.

Then the Lyapunov function $V : x \longrightarrow x^t P x$ is obtained with

$$P = \begin{bmatrix} 2.01656529 & 0.08342414 & 0.11486412 & -1.29171207 \\ 0.08342414 & 0.07182875 & -0.08901516 & -0.04613902 \\ 0.11486412 & -0.08901516 & 1.35166103 & -0.56806025 \\ -1.29171207 & -0.04613902 & -0.56806025 & 2.44792802 \end{bmatrix}.$$

The eigen values of P are 3.65640661, 1.40539696, 0.76532165 and 0.06085787. So the matrix is definitely defined positive.

5 SIMULATION

In order to simulate the different dynamics, I used an OMNotebook notebook and a Python code. I have entered some of the code in the appendix. The simulation begins near the equilibrium at for $z = 0.1$, $v_z = 0$, $u_1 = 0.0858$ and $u_2 = 0$.

5.1 Simulation of the dynamics

Before looking at the Lyapunov candidate for the initial system, we can first see if the linearized system well defines the initial system.

I implemented a Python code to simulate each dynamics and used the `matplotlib` librairy to plot the result. We can see in 7 and 8 that the linearization of the system is legitimate.

5.2 The Lyapunov function in the linearized and intial dynamics

We can see in 9 that the computed Lyapunov function can check the conditions to describe the stability of the linearized system.

Hopping this function is also well suited for the initial system describing the altitude controller, the figure 10 effectively suggests that the function is a candidate.

```python
import numpy as np
from numpy import linalg as LA
from control.matlab import *

''' CONSTANTS OF THE SYSTEM '''
g = 9.81
m = 0.028
k_1 = 2.130295*10**(-11)
k_2 = 1.032633*10**(-6)
k_3 = 5.484560*10**(-4)
scale = 1000
base = 36000
K_pz = 2
K_iz = 0.5
K_pv = 25
K_iv = 15
K_p = K_pv*scale
K_i = K_iv*scale
# the equilibrium PWM and u_1
delta = (4*k_2)**2 - 4*(4*k_1)*(4*k_3-m*g)
pwm_e = (-4*k_2 + np.sqrt(delta))/(2*4*k_1)
u_1_e = (pwm_e - base)/K_i
alpha = 8*pwm_e*k_1/m + 4*k_2/m

'''
We have an equation form dx/dt = Ax. P = lyap(A, Q)
     solves AP+PA^t=-Q but I want to solve A^tP+PA=-Q,
     then I use the transpose of A named A_T.
'''
Q = np.eye(4)
A_T = np.array([[0,     -2*K_p*alpha,    -2,   -1],
                [1,      -K_p*alpha,     -1,    0],
                [0,       K_i*alpha,      0,    0],
                [0,    0.5*K_p*alpha,   0.5,    0]])

P = lyap(A_T,Q) # solving the Lyapunov equation for Q=Id
print(LA.eig(P)) # checking if P is defined positive
print((A_T).dot(P)+P.dot((A_T).T)) # checking the solver
```

Figure 6: Python code to compute a Lyapunov function for the linearized system

6 PROOF OF THE LYAPUNOV CANDIDATE

6.1 An SOS problem

In order to prove that the Lyapunov candidate is a Lyapunov function, we used a free MATLAB toolbox SOSTOOLS [A. Papachristodoulou and Parrilo 2013].

SOSTOOLS makes it possible to formulate and solve sums of squares (SOS) optimization programs. First, a SOS polynomial is a polynomial that is a sum of squares of polynomials: $h(x) = \sum g_i(x)^2$. Second, a SOS optimization problem can be expressed as minimizing the function $z = c_1 * u_1 + ... + c_n * u_n$ subjects to the constraint $\forall i \in \{0, ..., N\}, P_i(x) := A_{i0}(x) + A_{i1}(x) * u_1 + ... + A_{in}(x) * u_n \in SOS$.

The connection with the Lyapunov candidate? As a reminder: $V : x \longrightarrow x^t P x$ with

$$P = \begin{bmatrix} 2.01656529 & 0.08342414 & 0.11486412 & -1.29171207 \\ 0.08342414 & 0.07182875 & -0.08901516 & -0.04613902 \\ 0.11486412 & -0.08901516 & 1.35166103 & -0.56806025 \\ -1.29171207 & -0.04613902 & -0.56806025 & 2.44792802 \end{bmatrix}$$

defined positive. Then this candidate is polynomial of order 2. We can use SOSTOOLS to prove the Lyapunov Theorem condition over

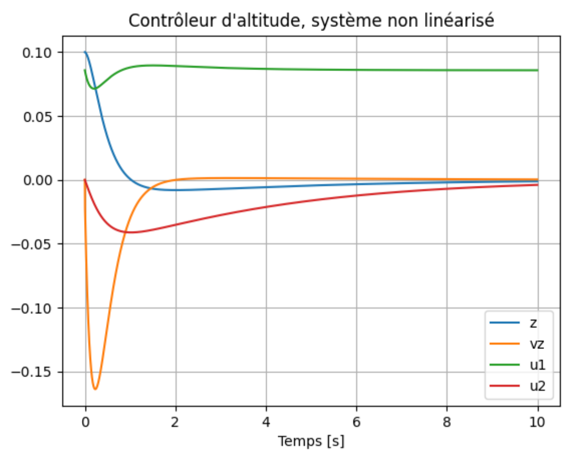


Figure 7: Python simulation of the initial system

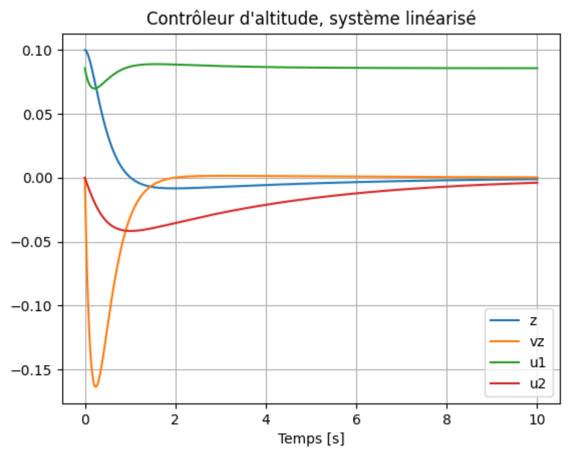


Figure 8: Python simulation of the linearized system

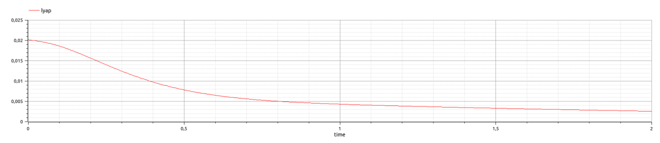


Figure 9: Lyapunov function simulated in the linearized system dynamics

$-\dot{V}$: if $-\dot{V}$ is a SOS then $-\dot{V}$ is positive. As $V(x) > 0$ for every state x of the space (even the unreachable) by definition of P, then V would be a Lyapunov function and the system would be stable.

However, the candidate is not a Lyapunov function for the entire space. It is not surprising, some states are not even reachable and the Lyapunov function is specific to the studied system and its bound. So we added bounding constraints. We defined SOS polynomials β as "variables" for the SOSTOOLS solver,

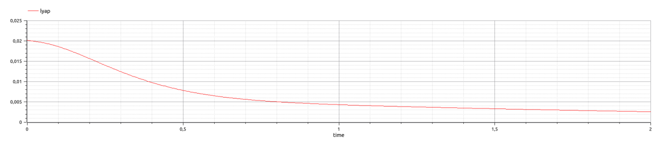


Figure 10: Lyapunov function simulated in the initial system dynamics

such that $-\dot{V}(x) = \beta_{1inf}(x) * (z - c_{1inf}) + [...] + \beta_{4sup}(x) * (c_{4sup} - u_2)$. If such β are found then $-\dot{V}(x) > 0$ for all $x \in \{[z\ v_z\ u_1\ u_2], \begin{cases} z \in [c_{1inf}, c_{1sup}] & u_1 \in [c_{3inf}, c_{3sup}] \\ v_z \in [c_{2inf}, c_{2sup}] & u_2 \in [c_{4inf}, c_{4sup}] \end{cases}\}$.

This is illustrated in the sequence of the MATLAB program in the appendix.

We now need to ensure that the chosen bounding contains all reachable states.

6.2 Reachability analysis

In order to found a bounding that ensure a rigorous proof of the system containment, we used Flow* previously presented in the beginning of this report.

I first simulated with arbitrary initial conditions on the altitude in order to have a hint of bounds. Then I declared `unsafe` conditions in order to be informed when states are out of bounds. And my first guess was good, it did not meet the `unsafe` set:

```
init
{z in [-0.1, 0.1]        vz in [0,0]
 u1 in [0.0857,0.0858] u2 in [0,0]
 t in [0,0]              g in [9.80, 9.81]}


unsafe
{z <= -0.15    z >= 0.15
 vz <= -0.3  vz >= 0.3
 u1 <= -0.08 u1 >= 0.1
 u2 <= -0.05 u2 >= 0.05}
```

Figures 11, 12, 13 and 14 show the reachability plots computed by Flow*.

I can now use those bounds to check the Lyapunov candidate on MATLAB.

6.3 Checking the candidate

The bounds inputted in the MATLAB program, we can see that the SOSTOOLS solver indeed finds SOS polynomials β such that $-\dot{V}(x) = \beta_{1inf}(x) * (z - c_{1inf}) + [...] + \beta_{4sup}(x) * (c_{4sup} - u_2)$.

The output of the solver is:

```
Residual norm: 5.4687e-10


         iter: 15
    feasratio: 1.0000
         pinf: 0
         dinf: 0
       numerr: 0
```

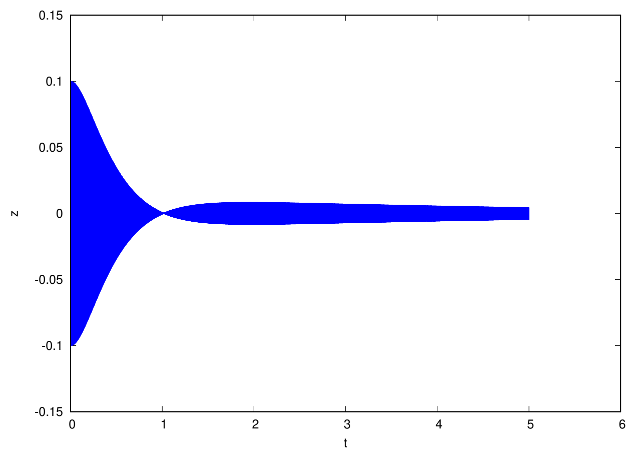


Figure 11: Flow* simulation of the drone's altitude

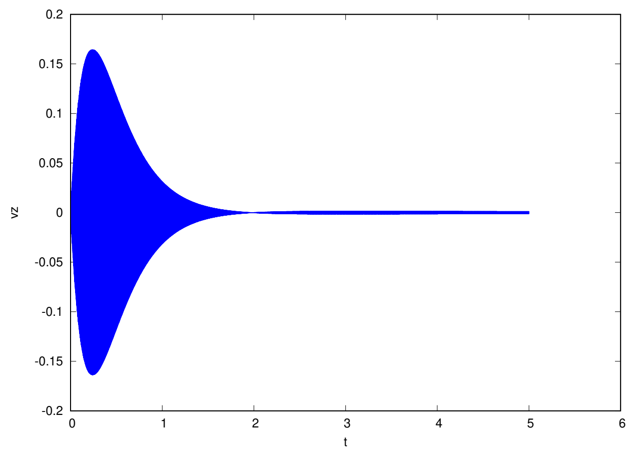


Figure 12: Flow* simulation of the drone's vertical velocity

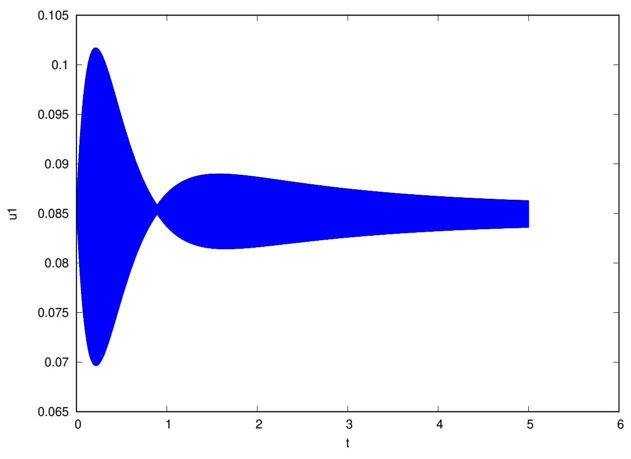


Figure 13: Flow* simulation of the state u_1

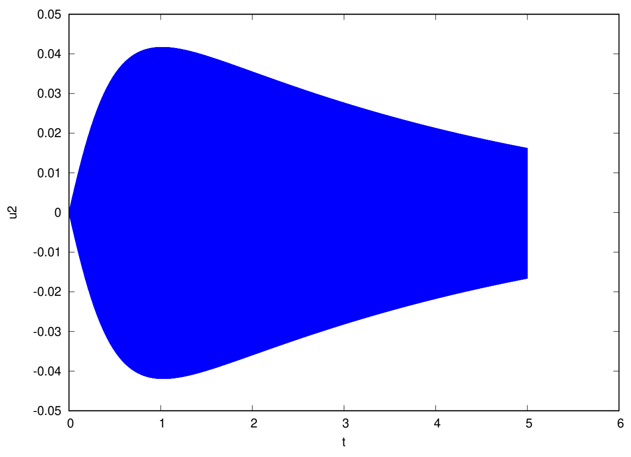


Figure 14: Flow* simulation of the state u_2

```
      r0:  1.1837e-11
  timing:  [0.1272  0.1862  0.0099]
 wallsec:  0.3233
  cpusec:  0.2700
```

It indicates that the precision is good enough, the `Residual norm` is way smaller than the incertitude on the matrix P (which is, as I tested, robust over perturbations).

We can now conclude that the Lyapunov candidate is a Lyapunov function and that the system is stable for an initial state near the equilibrium.

7 CONCLUSION

7.1 Summary

In this report, we have seen how to characterize a stability of a system, modelled the Crazyflie's altitude controller, simulated this altitude controller, linearized this system to simplify it, found a Lyapunov function for the linearized system and a candidate for the altitude controller system. We have proved the Lyapunov candidate to be a Lyapunov function.

7.2 Overview of the method

The general approach followed to formally verify the stability of the controller can be partitioned into sub-tasks:

- model the controller in order to form a system
- linearize the system around the equilibrium of the system
- compute a Lyapunov function for the linearize system by solving the Lyapunov equation
- use this Lyapunov function as candidate for the non-linear system
- bound the initial system to a neighborhood of the equilibrium (this neighborhood is the domain of use of the controller) with reachability analysis
- prove Lyapunov conditions on the bounded initial system using SOSTOOLS
- the next step of this formal verification method would be to insert this proof in the firmware code of the Crazyflie.

7.3 Next step: insertion of the proof in the code

To complete the formal verification of the Crazyflie's altitude controller system, we would need to include this proof in the SPARK code of the Crazyflie. The control system of this drone has also been translated into SPARK. SPARK is a formally defined language based on Ada. It is used for the development of critical systems. It ensure safety and security. To prove programs correctness we can use GNATprove, a tool of the Ada/SPARK compiler GNAT (from GCC). GNATprove uses the Why3 intermediate language for deductive program verification. Then the next step would be to translate our proof of stability in Why3.

ACKNOWLEDGMENTS

To Sylvie Putot, Eric Goubault, Alexandre Chapoutot, Christophe Garion and Khalil Ghorbal for their help and support.

REFERENCES

G. Valmorbida S. Prajna P. Seiler A. Papachristodoulou, J. Anderson and P. A. Parrilo. 2013. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB.* http://arxiv.org/abs/1310.4716. Available from http://www.eng.ox.ac.uk/control/sostools, http://www.cds.caltech.edu/sostools and http://www.mit.edu/~parrilo/sostools.
BitCraze. 2020. *Crazyflie 2.0.* https://www.bitcraze.io/documentation/repository/.
J. Förster. 2015. System Identification of the Crazyflie 2.0 Nano Quadrocopter. (2015).
E. Goubault S. Putot. 2019. Inner and Outer Reachability for the Verification of ControlSystems. (2019).
Erika Ábrahám Xin Chen, Sriram Sankaranarayanan. 2017. *Flow**. Available from https://flowstar.org/.

A CRAZYFLIE FIRMWARE CODE

A.1 Crazyflie 1.0 firmware code including altitude control

Code found in Bitcraze GitHub repository: https://github.com/bitcraze/crazyflie-firmware/blob/master/src/modules/src/position_controller_pid.c

```c
static const float thrustScale = 1000.0f;

[...]

static struct this_s this = {

  [...]

  .pidVZ = {
    .init = {
      .kp = 25,
      .ki = 15,
      .kd = 0,
    },
    .pid.dt = DT,
  },

  [...]

  .pidZ = {
    .init = {
      .kp = 2.0f,
      .ki = 0.5,
      .kd = 0,
    },
    .pid.dt = DT,
  },

  .thrustBase = 36000,
  .thrustMin  = 20000,
};
```

```c
void positionController(float* thrust, attitude_t *
    attitude, setpoint_t *setpoint,

        const state_t *state)
{
  this.pidX.pid.outputLimit = xyVelMax * velMaxOverhead;
  this.pidY.pid.outputLimit = xyVelMax * velMaxOverhead;
  // The ROS landing detector will prematurely trip if
  // this value is below 0.5
  this.pidZ.pid.outputLimit = fmaxf(zVelMax, 0.5f)  *
    velMaxOverhead;

  float cosyaw = cosf(state->attitude.yaw * (float)M_PI /
     180.0f);
  float sinyaw = sinf(state->attitude.yaw * (float)M_PI /
     180.0f);
  float bodyvx = setpoint->velocity.x;
  float bodyvy = setpoint->velocity.y;

  // X, Y
  if (setpoint->mode.x == modeAbs) {
    setpoint->velocity.x = runPid(state->position.x, &
      this.pidX, setpoint->position.x, DT);
  } else if (setpoint->velocity_body) {
    setpoint->velocity.x = bodyvx * cosyaw - bodyvy *
      sinyaw;
  }
  if (setpoint->mode.y == modeAbs) {
    setpoint->velocity.y = runPid(state->position.y, &
      this.pidY, setpoint->position.y, DT);
  } else if (setpoint->velocity_body) {
    setpoint->velocity.y = bodyvy * cosyaw + bodyvx *
      sinyaw;
  }
  if (setpoint->mode.z == modeAbs) {
    setpoint->velocity.z = runPid(state->position.z, &
      this.pidZ, setpoint->position.z, DT);
  }

  velocityController(thrust, attitude, setpoint, state);
}

void velocityController(float* thrust, attitude_t *
    attitude, setpoint_t *setpoint,

        const state_t *state)
{
  this.pidVX.pid.outputLimit = rpLimit * rpLimitOverhead;
  this.pidVY.pid.outputLimit = rpLimit * rpLimitOverhead;
  // Set the output limit to the maximum thrust range
  this.pidVZ.pid.outputLimit = (UINT16_MAX / 2 /
    thrustScale);
  //this.pidVZ.pid.outputLimit = (this.thrustBase - this.
    thrustMin) / thrustScale;

  // Roll and Pitch
  float rollRaw  = runPid(state->velocity.x, &this.pidVX,
     setpoint->velocity.x, DT);
  float pitchRaw = runPid(state->velocity.y, &this.pidVY,
     setpoint->velocity.y, DT);

  float yawRad = state->attitude.yaw * (float)M_PI / 180;
  attitude->pitch = -(rollRaw  * cosf(yawRad)) - (
    pitchRaw * sinf(yawRad));
  attitude->roll  = -(pitchRaw * cosf(yawRad)) + (rollRaw
      * sinf(yawRad));

  attitude->roll  = constrain(attitude->roll,  -rpLimit,
    rpLimit);
  attitude->pitch = constrain(attitude->pitch, -rpLimit,
    rpLimit);

  // Thrust
  float thrustRaw = runPid(state->velocity.z, &this.pidVZ
    , setpoint->velocity.z, DT);
  // Scale the thrust and add feed forward term
  *thrust = thrustRaw*thrustScale + this.thrustBase;
  // Check for minimum thrust
  if (*thrust < this.thrustMin) {
    *thrust = this.thrustMin;
  }
}
```

A.2 Analysis of the use of the thrust value in the firmware code

I tried to find in the code the way in which was used `thrust`.

In the file `controller_pid.c`, the parameter `*control` of the `controllerPid(...)` function contains `actuatorThrust` computed in `positionController(...)` that I presented above (it computes this thrust with the double PI controller):

```
void controllerPid(control_t *control,  setpoint_t *
     setpoint,
                                        const
     sensorData_t *sensors,
                                        const state_t *
     state,
                                        const uint32_t
     tick)
{
    [...]

    positionController(&actuatorThrust, &attitudeDesired,
      setpoint, state);

    [...]

    control->thrust = actuatorThrust;

    [...]
}
```

The function `controllerPid(...)` is applied in the function `controller(...)` defined in `controller.c`. It takes as parameter the `*control` parameter that interests us:

```
static ControllerFcns controllerFunctions[] = {
  {.init = 0, .test = 0, .update = 0, .name = "None"}, //
     Any
  {.init = controllerPidInit, .test = controllerPidTest,
    .update = controllerPid, .name = "PID"},

  [...]
};

[...]

void controller(control_t *control, setpoint_t *setpoint,
      const sensorData_t *sensors, const state_t *state,
     const uint32_t tick) {
  controllerFunctions[currentController].update(control,
    setpoint, sensors, state, tick);
}
```

It is in `stabilizer.c` that we can find a use of `controller(...)` as well as what seems to me to be the first occurence of `control`. It is the one that interests me because it is used as it comes out of `controller(...)` in another function that we have to find in the files named `powerDistribution(&control)`:

```
/* The stabilizer loop runs at 1kHz (stock) or 500Hz (
     kalman). It is the
 * responsibility of the different functions to run
     slower by skipping call
 * (ie. returning without modifying the output structure)
     .
 */

static void stabilizerTask(void* param)
{
     [...]

     controller(&control, &setpoint, &sensorData, &state,
      tick);

     checkEmergencyStopTimeout();

     checkStops = systemIsArmed();
     if (emergencyStop || (systemIsArmed() == false)) {
          powerStop();
     } else {
          powerDistribution(&control);
     }

     [...]
}
```

And so finally the value `thrust` is used directly as PWM for the motors in the case of a simple altitude control. We can see that in the following file `power_distribution_stock.c`:

```
#define limitThrust(VAL) limitUint16(VAL)

[...]

void powerDistribution(const control_t *control)
{
  #ifdef QUAD_FORMATION_X
    int16_t r = control->roll / 2.0f;
    int16_t p = control->pitch / 2.0f;
    motorPower.m1 = limitThrust(control->thrust - r + p +
      control->yaw);
    motorPower.m2 = limitThrust(control->thrust - r - p -
      control->yaw);
    motorPower.m3 =  limitThrust(control->thrust + r - p
     + control->yaw);
    motorPower.m4 =  limitThrust(control->thrust + r + p
     - control->yaw);
  #else // QUAD_FORMATION_NORMAL
    motorPower.m1 = limitThrust(control->thrust + control
     ->pitch +
                                  control->yaw);
    motorPower.m2 = limitThrust(control->thrust - control
     ->roll -
                                  control->yaw);
    motorPower.m3 =  limitThrust(control->thrust -
     control->pitch +
                                  control->yaw);
    motorPower.m4 =  limitThrust(control->thrust +
     control->roll -
                                  control->yaw);
  #endif

  [...]
```

B PYTHON SIMULATIONS CODE

B.1 Initial dynamics

```python
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
[...]
def f(x, pwm):
  """
  Avec x un vecteur contenant :
  - z   : l'altitude du dr ne
  - v_z : la vitesse verticale du dr ne
  - u1  : correspondant   \int(-2z+0.5*\int(-z)-v_z)
  - u2  : correspondant   \int(-z)
  """
  dz = x[1]
  dv_z = 4/m*(k_1*pwm**2 + k_2*pwm + k_3) - g
  du_1 = -2*x[0] + 0.5*x[3] -x[1]
  du_2 = -x[0]

  dx = np.array([dz, dv_z, du_1, du_2])

  return dx

def EulerMethod(f, x0, t0, tend, h):
    T = np.arange(t0, tend, h)
    X = [0 for i in T]
    x = x0
    pwm = 0

    i = 0
    for t in T :
        X[i] = x
        dx = f(x, pwm)
        x = x + h*dx
        pwm = K_p*dx[2] + K_i*x[2] + base
        i += 1

    X = np.array(X)
    plt.plot(T, X[:,0], label='z')
    [...]
    plt.show()

EulerMethod(f, np.array([1,0,u_1_e,0]), 0, 10, 0.002)
```

B.2 Linearized dynamics

```python
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
[...]
def f(Dx):
  """
  Avec Dx un vecteur contenant :
  - Dz   :  cart    l'  quilibre   de l'altitude du dr ne
  - Dv_z :  cart    l'  quilibre   de la vitesse verticale du dr ne
  - Du1  :  cart    l'  quilibre   de \int(-2z+0.5*\int(-z)-v_z)
  - Du2  :  cart    l'  quilibre   de \int(-z)
  """
  return A.dot(Dx)

def EulerMethod(f, x0, t0, tend, h):
    T = np.arange(t0, tend, h)
    X = [0 for i in T]
    x = x0
    i = 0
    for t in T :
      X[i] = x
      x = x + h*f(x)
      i += 1
    X = np.array(X)

    for j in range(i) :
      X[j,2] = X[j,2] + u_1_e

    plt.plot(T, X[:,0], label='z')
    [...]
    plt.show()

EulerMethod(f, np.array([1,0,0,0]), 0, 10, 0.0002)
```

C OMNOTEBOOK CODE

Code written in the OMNotebook in order to simulate the different dynamics and functions in the dynamic of the systems.

C.1 Altitude controller definition in OMNotebook

```
class Dynamique_nonLineaire
  Real z(start = 0.1,fixed=true);
  Real vz(fixed=true);
  Real u1(start = 0.0858,fixed=true);
  Real u2(fixed=true);
  Real pwm();
equation
  der(u2) = -z;
  der(z) = vz;
  pwm = 25000*der(u1) + 15000*u1 + 36000;
  der(vz) = 4/0.028*(2.130295*10^(-11)*pwm
      *pwm+1.032633*10^(-6)*pwm
      +5.484560*10^(-4))-9.81;
  der(u1) = -2*z+0.5*u2-vz;
end Dynamique_nonLineaire;
```

C.2 Linearized system definition in OMNotebook

```
class Dynamique_Lineaire
  Real z(start = 0.1,fixed=true);
  Real vz(fixed=true);
  Real u1(fixed=true);
  Real u2(fixed=true);
equation
  der(z)  = vz;
  der(vz) = -18.771099*z-9.3855495*vz
      +5.6313297*u1+4.69277475*u2;
  der(u1) = -2*z-vz+0.5*u2;
  der(u2) = -z;
end Dynamique_Lineaire;
```

C.3 Definition of the Lyapunov function in the linearized system

```
class Dynamique_Lineaire_Lyapunov
    Real z(start = 0.1,fixed=true);
    Real vz(fixed=true);
    Real u1(fixed=true);
    Real u2(fixed=true);
    Real lyap();
equation
    der(z)  = vz;
    der(vz) = -18.771099*z-9.3855495*vz
        +5.6313297*u1+4.69277475*u2;
    der(u1) = -2*z-vz+0.5*u2;
    der(u2) = -z;
    lyap = 2.01656529*z^2 + 0.16684828*z*vz
        + 0.22972824*z*u1 - 2.58342414*z*u2 +
         0.07182875*vz^2 - 0.17803032*vz*u1 -
         0.09227804*vz*u2 + 1.35166103*u1^2 -
         1.1361205*u1*u2 + 2.44792802*u2^2;
end Dynamique_Lineaire_Lyapunov;
```

C.4 Definition of the Lyapunov function in the initial system

```
class
    Dynamique_nonLineaire_Lyapunov_SOSTOOLS
    Real z(start = 0.1,fixed=true);
    Real vz(fixed=true);
    Real u1(start = 0.0858,fixed=true);
    Real u2(fixed=true);
    Real pwm();
    Real lyap();
equation
    der(u2) = -z;
    der(z) = vz;
    pwm = 25000*der(u1) + 15000*u1 + 36000;
    der(vz) = 4/0.028*(2.130295*10^(-11)*pwm
        ^2+1.032633*10^(-6)*pwm
        +5.484560*10^(-4))-9.81;
    der(u1) = -2*z+0.5*u2-vz;
    lyap =- 2.016e-16*u1^2 + 1.499e-16*u1*u2
         - 2.101e-17*u1*vz - 4.076e-17*u1*z +
         0.5894*u2^2 + 0.04008*u2*vz -
        0.04917*u2*z + 0.6629*vz^2 -
        0.0009188*vz*z + 0.6648*z^2;
end
    Dynamique_nonLineaire_Lyapunov_SOSTOOLS
    ;
```

D FLOW* MODEL

```
continuous reachability
{state var z, vz, u1, u2, t, g

 par
 {k1 = 2.130295e-11   k2 = 1.032633e-6
  k3 = 5.484560e-4    Kp = 25000
  Ki = 15000          m = 0.028
  base = 36000}

 setting
 {fixed steps 0.003
  time 5
  remainder estimation 1e-1
  identity precondition
  gnuplot octagon t, z
  fixed orders 5
  cutoff 1e-8
  precision 100
  output drone
  print on}

 poly ode 1 { 200 }
 {z' = vz
  vz' = 142.857142857 * (k1 * (Kp * (-2 * z
      + 0.5 * u2 - vz) + Ki * u1 + base) ^ 2
       + k2 * (Kp * (-2 * z + 0.5 * u2 - vz)
       + Ki * u1 + base) + k3) - g
  u1' = -2*z+0.5*u2-vz
  u2' = -z

  t' = 1  g' = 0}

 init
 {z in [-0.1, 0.1]       vz in [0,0]
  u1 in [0.0857,0.0858]  u2 in [0,0]
  t in [0,0]             g in [9.80, 9.81]}}

unsafe
{z <= -0.15     z >= 0.15
 vz <= -0.3   vz >= 0.3
 u1 <= -0.08  u1 >= 0.1
 u2 <= -0.05  u2 >= 0.05}
```

E MATLAB PROGRAM

E.1 MATLAB program sequence to check if the candidate is a Lyapunov function using SOSTOOLS

```
% MATLAB program to check if the Lyapunov
% candidate found is a Lyapunov function
[...]
```

```

% ========================================
% Constructing the model:
g = 9.81;
m = 0.028;
k_1 = 2.130295*10^(-11);
k_2 = 1.032633*10^(-6);
k_3 = 5.484560*10^(-4);
scale = 1000;
base = 36000;
K_pz = 2;
K_iz = 0.5;
K_pv = 25;
K_iv = 15;
K_p = K_pv*scale;
K_i = K_iv*scale;

delta = (4*k_2)^2 - 16*k_1*(4*k_3-m*g);
pwm_e = (-4*k_2 + sqrt(delta))/(8*k_1);
u_1_e = (pwm_e-base)/K_i;

% ========================================
% Constructing the vector field dx/dt = f:
f = [- z;
     vz;
     4/0.028*(k_1*(K_p*(-K_pz*z+K_iz*u2-vz) +
          K_i*u1 + base)^2+k_2*(K_p*(-K_pz*z+
        K_iz*u2-vz) + K_i*u1 + base)+k_3)-g;
     -K_pz*z+K_iz*u2-vz];
[...]

% ========================================
% Next, define the Lyapunov candidate:
V = 2.01656529*z^2 + 0.16684828*z*vz +
    0.22972824*z*u1 - 2.58342414*z*u2 -
    0.0197097015678043*z + 0.07182875*vz^2 -
     0.17803032*vz*u1 - 0.09227804*vz*u2 +
    0.0152742408909792*vz + 1.35166103*u1^2
    - 1.1361205*u1*u2 - 0.231933483860155*u1
     + 2.44792802*u2^2 + 0.0974742852688227*
    u2 + 0.00994945103498116;
dV = diff(V,z)*f(1)+diff(V,vz)*f(2)+diff(V,
    u1)*f(3)+diff(V,u2)*f(4);

% ========================================
% Bounders:
c_1_inf = -0.15; c_2_inf = -0.3; c_3_inf =
    -0.08; c_4_inf = -0.05;
c_1_sup = 0.15; c_2_sup = 0.3; c_3_sup =
    0.1; c_4_sup = 0.05;

% ========================================
% SoS polynomials beta:
[prog,beta_1_inf] = sospolyvar(prog,
    monomials(vars,[0,1,2]), 'wscoeff');
[...]
[prog,beta_4_sup] = sospolyvar(prog,
    monomials(vars,[0,1,2]), 'wscoeff');

% ========================================
% Next, define SOSP constraints

% Constraint 1: V(x) >= 0
prog = sosineq(prog,V);
% Constraint 2: -dV/dt(x) = beta_1_inf(x)*(z
    - c_1_inf) + [...] + beta_4_sup(x)*(
    c_4_sup - u2)
prog = soseq(prog,dV + beta_1_inf*(z -
    c_1_inf) + [...] + beta_4_sup*(c_4_sup -
     u2));
[...]
```