Height regulation of a quadrotor helicopter

Paolo Forni, Fabio Garofalo, Federico Patota, Flavio Tonelli, Luca Veroli
Università degli Studi di Roma "La Sapienza"
(Dated: May 22, 2012)

***Abstract* - Quadrotor helicopters represent a thrilling and significant field of research for control engineers and this is why we focused on such of aerial vehicles. This paper describes the development of a height control system which involved the identification of the dynamic model, the design and implementation of the controller. We managed to work out the several control-related issues we dealt with and this led us to some stimulating conclusions.**

1. INTRODUCTION

Quadrotors and in general unmanned aerial vehicles (UAVs) are gaining increasing interest because of a wide area of applications. They are commonly employed for military purposes, academic research and commercial applications.

The aim of the current work is the design and implementation of a height regulation system on a quadrotor helicopter based on standard control techniques and the use of the Arduino prototyping platform and of a simple sonar. For technical and economical reasons we addressed this topic just focusing on the height regulation assuming that a complete attitude control was already provided. This resulted as the most feasible and less expensive way to get through this work.

The choice of a small-scale model capable of meeting the challenges this kind of job would present turned out to be essential. GAUI 330X small-scale model device resulted as the best choice fitting with our needs: actually, it provides high stability thanks to the GU-344 three-axis gyro stabilizing system.

We assembled the main quadrotor body frame and then we passed through the following steps:

- we wired and installed Electronic Speed Controllers (ESCs), motors, the control unit and the receiver;
- we set the radio control and the control unit;
- we tuned the ESCs to make sure each motor is run with the same output signal;
- we set the ESCs to select brake mode, start mode and other similar parameters;
- we calibrated the neutral point before the first flight (the so-called *trig* part).

In order to obtain significant measurements and to interface the control unit with our own controller, we realized an electronic scheme endowed with the Arduino

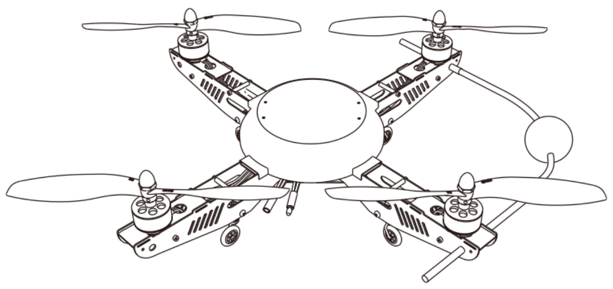
FIG. 1: CAD image of the GAUI 330X small-scale model

Nano v3.0 prototyping platform, the Devantech SRF05 ultra-sonic ranger and a potentiometer. At the end of the work we realized the final electronic scheme (see appendix A) from which we removed the potentiometer.

Each step of this work required an intensive debugging phase especially regarding the coding part of the Arduino board. The final sketch is reported in the appendix B.

2. DYNAMIC MODEL

2.1. The linearized dynamic model

The general dynamic model of a quadrotor is well known in the literature and will not be discussed here in all details. An extensive model refers to each angular velocity ω_{m_i} given to motor i (and so to the propellers) as input to a MIMO nonlinear system with 12 state variables, defined as follows:

$$u = \begin{pmatrix} \omega_{m_1} \\ \omega_{m_2} \\ \omega_{m_3} \\ \omega_{m_4} \end{pmatrix} \qquad x = \begin{pmatrix} x_0 \\ y_0 \\ h \\ \varphi \\ \theta \\ \psi \\ v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

where $\begin{pmatrix} x_0 & y_0 & h \end{pmatrix}^T$ is the position vector, φ, θ, ψ are the roll-pitch-yaw variables, $\begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^T$ is the velocity vector and $\begin{pmatrix} \omega_x & \omega_y & \omega_z \end{pmatrix}^T$ is the angular velocity vector.

This model takes into account air damping forces, gravity, each motor thrust and non-modelled disturbances.

Fortunately, the GU-344 three-axis gyro system already provided an attitude control and for this reason we focused on the height regulation. The control unit hides the four angular velocity commands u but it gives us four new commands which are the well known flight dynamics parameters: *throttle*, *aileron*, *elevator* and *rudder*.

Starting from this configuration we built up an *ad hoc* model from scratch. We reduced the problem to the neighborhood of an equilibrium point where we assumed the quadrotor fully stabilized in attitude, i.e. $\left(\bar{\varphi}\ \ \bar{\theta}\ \ \bar{\psi}\ \ \bar{v_x}\ \ \bar{v_y}\ \ \bar{v_z}\ \ \bar{\omega_x}\ \ \bar{\omega_y}\ \ \bar{\omega_z}\right)^T = \mathbf{0}$. Note that in such configuration the ranger would give us the correct height measurement even if the quadrotor was slightly tilted and this is due to the fact that the sonar beam has a wide cone angle.

Within this we decided to control the height ignoring the *aileron*, *elevator* and *rudder* commands. Actually, the *throttle* command τ is associated with the overall average force F_p given by propellers to the rigid body directly causing the raise or the descent. Thus, the model was reduced to a SISO system consisting of the *throttle* command τ as input and the height h as the only output.

We are now going to explain the relation between F_p and τ. The *throttle* command is a Pulse-Position Modulation (PPM) signal to the control unit and we define τ as the time length (expressed in microseconds) of high state of the square wave. The GAUI instruction manual reports that the *throttle* command is supposed to be proportional to the overall angular velocity ω of the propellers and this is due to the internal ESC control loop: $\omega = k'\tau$ where k' is a generic positive proportionality constant. Moreover, the relation between the overall thrust force F_p of the propellers and ω is given by aerodynamics: $F_p = b_p\,\omega^2$ where b_p is a coefficient that depends on the propeller material and shape. Albeit we cannot determine the exact expression of the relationship between F_p and τ we can assume $F_p(\tau)$ a monotonically increasing function of τ. Linearizing, we obtain:

$$\Delta F_p(\tau) = k\Delta\tau$$

with k a generic positive proportionality constant.

The simplifications adopted so far lead to the formulation of our own model. Given Newton's first law:

$$m\ddot{\mathbf{x}} = \sum_i \mathbf{F_i}$$

in one dimension we have:

$$m\ddot{h} = F_p(\tau) - mg - b_v\dot{h}$$

where $b_v\dot{h}$ is the air damping force on the vertical axis and b_v is the damping coefficient which has been experimentally determined.

Setting $\gamma_v = \frac{b_v}{m}$ and $A_p(\tau) = \frac{F_p(\tau)}{m}$ we have the nonlinear system:

$$\ddot{h} + \gamma_v\dot{h} = A_p(\tau) - g \qquad (1)$$

In the neighborhood of an equilibrium point $\left(\bar{\tau}, \bar{h}\right)$ where the thrust balances the gravity, i.e. $A_p(\bar{\tau}) = g$, we can consider the gravity a constant disturbance. This leads to the corresponding linearized system with the following transfer function:

$$W(s) = \frac{\Delta h(s)}{\Delta\omega(s)} = \frac{k}{s(s+\gamma_v)} \qquad (2)$$

2.2. Parameters definition and validation

The lack of knowledge about the exact nonlinear relation $A_p(\tau)$ in (1) forced us to base our analysis on the linearized model (2). Thus, we decided to take some measurement sessions to identify the process, assign values to all mathematical constant in (2) and find the exact relation between the input and the output. Our model is based on the unit-step response.

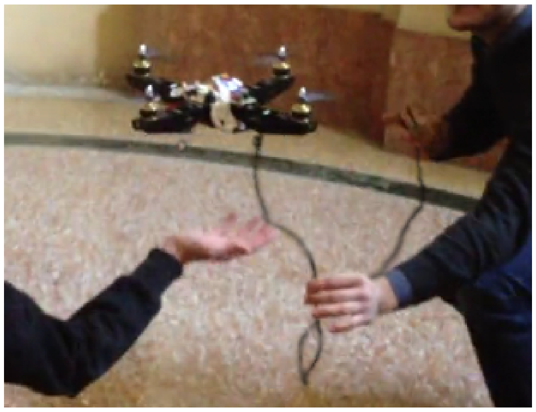

FIG. 2: Measurement session with the USB cable

For this purpose, we soldered an electric connection scheme (see appendix A) on a protoboard and coded an Arduino sketch (see appendix B) to record the following data: the *throttle* command τ to the control unit and the height h from the sonar.

- As a first attempt, we made Arduino send raw data through a USB cable as depicted in figure 2. Unfortunately, the cable destabilized the quadrotor in such a way that no interesting measures were obtained. Writing data on the internal Arduino EEPROM resulted to be the best choice despite the small amount of memory.

- As a second attempt, manually commanding the *throttle* thanks to an external potentiometer, we successfully recorded useful measures. We made the quadrotor take off and, once reached a good equilibrium point, we provided an unit-step to the *throttle* command. Then we turned off the quadrotor and downloaded raw data from the EEPROM thanks to an extremely simple Python script (see appendix C).

The real measured output provided by the sonar was then compared with the simulated output given by the Simulink model reported in D. We toggled the parameters k and γ_v in (2) to get the results showed in figures 3 and 4 and reported in table I. We finally chose the mean value $k = 2\,[\frac{10^4 m}{s^3}]$ and $\gamma_v = 0.2\,[s^{-1}]$ as the best fitting parameters.

Take n.	$k[\frac{10^4 m}{s^3}]$	$\gamma_v[s^{-1}]$
01	2.4322	0.2
02	1.6126	0.2

TABLE I: Best fitting parameters

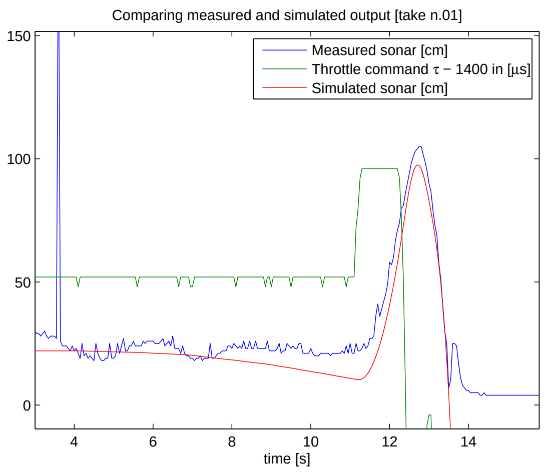

FIG. 3: Identification of parameters about the first data-set

3. CONTROLLER DESIGN

3.1. PI and PID controllers

Our goal was to design a height regulation for the quadrotor. The required specifications follow:

- **asymptotic stability** about a specified equilibrium point;
- **zero steady-state error** for unit step input.

Since the transfer function $W(s)$ of the process itself coitains a pole in $s = 0$, both the given specifications are satisfied by a control law based on a simple proportional action:

$$\tau(t) = K_P\, e(t) \tag{3}$$

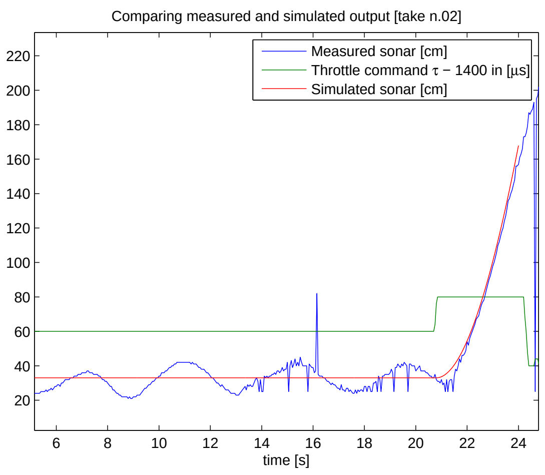

FIG. 4: Identification of parameters about the second data-set

where K_P is the proportional action constant and $e(t) = h_d(t) - h(t)$ is the difference between the desired height (setpoint) and the real height of the quadrotor. The linearized system given by 2 doesn't consider nonlinearities and inaccuracy of the mathematical model. Firstly, our controller doesn't act directly upon the ESCs but it is interfacing the control unit. Secondly, we have to take into account quantization error in τ and in the measured h. Thus, we added an integral action to the control law in order to deal with all these non-modelled dynamics, as follows:

$$\tau(t) = K_P\, e(t) + K_I \int_0^t e(x)\, dx \tag{4}$$

where K_I is the integral action constant. As it will be shown in subsection 3.2, such a control law produces a large number of oscillations and a large overshoot. Thus we added a derivative action to improve the combined controller-process stability and this is why we focused on standard PID controllers. The derivative action is highly sensitive to noise that comes from the sonar and for this reason we filtered the ideal derivative by a first-order system. The final form of the PID algorithm follows:

$$PID(s) = K_P + \frac{K_I}{s} + \frac{sK_D}{1 + \frac{K_D}{K_P N}s} \tag{5}$$

where K_D is the derivative action constant and N is a value set to achieve low-pass filtering.

We are now going to justify the use of a digital PID controller. Firstly, the output signal τ and the measured signal h are both digital. Secondly, the sonar frequency $(20[Hz])$ set the whole control loop at the same frequency and it is far higher than the quadrotor's one. In fact we have $W(j\,(20\,[Hz])) = -77.95\,[dB]$ from the Bode magnitude plot. We can see the quadrotor as a very slow system that acts as a filter upon the *throttle* command.

Moreover, we compute the derivate basing on the last three samples, instead of the last two, to avoid the amplification of noise signal in the measured height. The algorithm coded into the Arduino is the following:

$$\begin{cases} P_t = K_P\, e_t \\ I_t = I_{t-1} + K_I\, T_s\, e_t \\ D_t = \frac{K_D}{K_D + K_P N T_s} \left(D_{t-1} - K_P N \frac{h_t - h_{t-2}}{2} \right) \end{cases}$$

where $T_s = 0.05\,[ms]$ is the sample time and h_t is the measured height at time t.

3.2. Simulations

The Simulink model reported in appendix D let us verify the correctness of our algorithm and check the duration of the unit step response. The tuning of the PID parameters was done by the Matlab auto-tuning tool. Table II shows values of parameters used in the simulations. Results are showed in figure 5 and 6.

	K_P	K_I	K_D	N
PI	0.29232	0.01662	0	
PID	0.29672	0.01570	0.10581	3.59690

TABLE II: Best fitting parameters

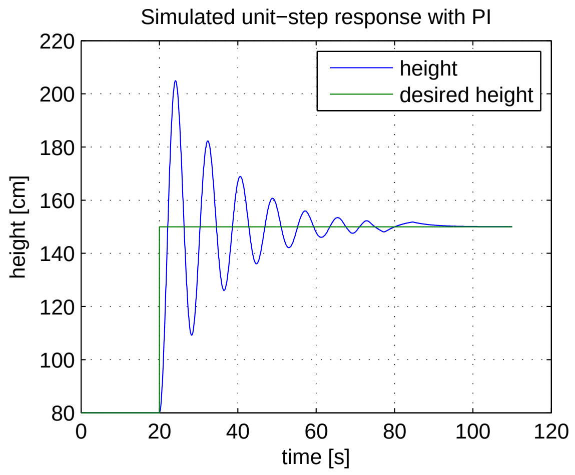

FIG. 5: Simulated unit-step response with PI

4. IMPLEMENTATION RESULTS

A really challenging part of this job was represented by the implementation of the control system in the quadrotor. We chose to make the quadrotor take off in manual mode and then to switch on our control system in order

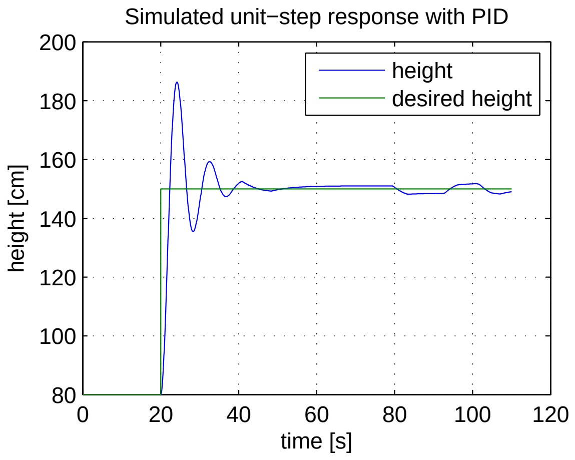

FIG. 6: Simulated unit-step response with PID

to start the *auto-throttle pilot* in the neighborhood of an equilibrium point.

Several problems occurred during the coding part of this *throttle* switch and this was due to the fact that high-level Arduino libraries don't handle timing problems in smart way. For instance, it wasn't possible to record raw data on the internal EEPROM while the control loop was running. Moreover, comparing the throttle square wave in manual and automatic mode thanks to an oscilloscope, we found a mismatch ($43\,[\mu s]$) in the time length of high state of the square wave.

About running standard PI controller, we observed a correct behaviour in the height regulation with small aerodynamic disturbances (about $30\,[cm]$ of absolute error). Then we provided a large unit-step input ($1\,[m]$) thanks to a wooden board. Too many oscillations (at least 6) occur when a large unit-step is provided. This is mainly due to the two integrators contained in the open-loop transfer function and to the amplification due to nonlinear effects. High K_P gain determine too large overshoots whereas low K_P gain make the overall system too slow and incapable of react properly within the environment constraints (floor, ceiling and the wooden board).

About running standard PID controller, we observed better performances in the height regulation with small aerodynamic disturbances (about $15\,[cm]$ of absolute error). Unfortunately, the system turns out to be extremely susceptible to too large unit-step input. This is mainly due to the fact that K_D si too high and we didn't simulate properly the first-order filter on derivative.

Finally, please note that we ignore the exact mapping between *throttle* and the overall force given to propellers by the GU-344 control unit. Actually, this control unit might add its own dynamics to the input *throttle* command and cause undesired behaviours.

5. CONCLUSIONS

This work was extremely interesting because let us:

- see how mathematical control laws effectively act upon the physical world;
- compare the simulation routine and the real implementation routine within a control-related context;
- deal with several technical issues.

Note that achieved performances are good but they can be largely improved by further works consisting in:

- increasing K_P gain to make the system more reactive;
- decreasing K_I gain to reduce the overshoots in oscillations;
- decreasing K_D gain to improve stability;
- using second-order filter on derivative to avoid noise amplification.

Appendix A: Electric schemes

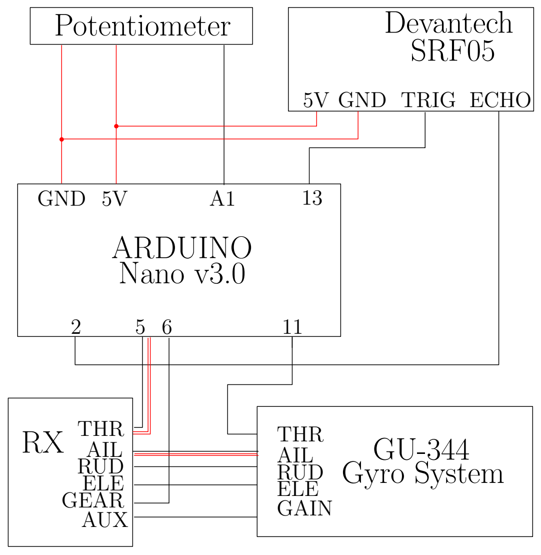

FIG. 7: Electrical scheme

Appendix B: Sketch Arduino

- **FLYDUINO_unit-step_eeprom.ino**

```
// Second measurement session:
// - the potentiometer commands the throttle;
// - throttle+sonar are recorded on the EEPROM.
//
// by Paolo Forni, Flavio Tonelli
// Feb 29, 2012

#include <Servo.h>
#include <EEPROM.h>

// Digital pins
#define TRIG 13
#define ECHO 2
int throttlePin = 11;
int potentiometerPin = 1;

// Servo objects
Servo throttle; //Orange

// INTERRUPT pins
#define INTERRUPT 0

// acqusition time intervals:
//  Sonar_Bw = 20 Hz
unsigned long sonarTimeInterval = 50000;
unsigned long trigTimeInterval = 13;

// all time variables are in microseconds
unsigned long startEchoTime=0;
unsigned long startSonarTime = 0;
unsigned long startTrigTime = 0;
unsigned long echoLength = 0;
unsigned long currentTime = 0;
unsigned long beginningTime = 0;

// state of handlers
int interruptState = 0;
int oldInterruptState=0;
int trig = 0;
int oldTrig = 0;
int gradino = 25;  // microseconds

// enable TRIG pin to be raised or lowered
int trigEnable = 1;
int firstChangingEdgeEvent = 0;

// eeprom values
int address = 0;
byte usValue = 0;
byte cmValue = 0;
boolean testEeprom = true;
boolean writeEepromFlag = false;
int recordThreshold = 1430;
int waitBeforeWriting = 0;
int waitIntervalEeprom = 20; // [sec]
int microseconds = 0;

// interrupt handler
void changeEdge() {
  if(firstChangingEdgeEvent==0)
// needed by the first
// cycle, otherwise it messes up everything
    firstChangingEdgeEvent=1;
  else
    interruptState = 1-interruptState;
}

void setMicroseconds(Servo &s, int uS) {
  if (s.readMicroseconds() != uS) {
    s.writeMicroseconds(uS);
  }
}


void setup() {
  // Serial.begin(9600);
  // pin initialization
  pinMode(TRIG,OUTPUT);
  throttle.attach(throttlePin);

  // attach interrupt
  attachInterrupt(INTERRUPT,changeEdge,CHANGE);

  // set some variables
  startSonarTime = micros();

}

void loop() {
  // takes currentTime in microseconds
  currentTime = micros();

  // sonar acquisition every sonarTimeInterval
  if(trigEnable==1&&
     currentTime-startSonarTime>=
     sonarTimeInterval) {

    trig = 1;
    trigEnable = 0;

    // zeroing time
    startSonarTime = currentTime;
  }

  // trig handler raising and lowering TRIG
  if(oldTrig!=trig) {
    oldTrig=trig;
    if(trig==1) {
      digitalWrite(TRIG,HIGH);
      startTrigTime = currentTime;
    }
    else {
      digitalWrite(TRIG,LOW);
    }
  }

  // interrupt handler sending echoLength
  if(oldInterruptState!=interruptState) {
    oldInterruptState=interruptState;
    if(interruptState==1) {
     startEchoTime = micros();
    }
    else {
     echoLength = (micros()-startEchoTime)/58;
     trigEnable = 1;
     int foo = analogRead(potentiometerPin);
     microseconds = map(foo,0,1023,1100,1900);
     setMicroseconds(throttle, microseconds);

     if(writeEepromFlag) {
      if(waitBeforeWriting>
         20*waitIntervalEeprom) {
       usValue = (microseconds-1100)/4;
       cmValue = echoLength;
       EEPROM.write(address,usValue);
       EEPROM.write(address+1,cmValue);
       address=address+2;
       if(address>1023) {
        writeEepromFlag=false;
//      Serial.print("RECORD OFF");
       }
      } else {
       waitBeforeWriting++;
      }
     }
    }
  }

  // let the TRIG pin high only during
  //trigTimeInterval
  if(trig==1&&currentTime-startTrigTime>=
     trigTimeInterval) {
    trig = 0;
  }

  if(microseconds>recordThreshold &&
     testEeprom) {
    testEeprom=false;
    writeEepromFlag=true;
//  Serial.println("RECORD ON");
    address=0;
  }

}
```

- **FLYDUINO_complete_PID.ino**

```
// FLYDUINO
// - can switch between automatic and manual
// - reads the sonar
// - provides throttle thanks to the PID
//
// by Paolo Forni, Flavio Tonelli,
// Apr 18, 2012

#include <PinChangeInt.h>
//extends interrupts to all pins
#include <TimerOne.h>
//simplifies timing on Timer1
#include <Servo.h> //simplifies servo control

// Digital pins and EXT INTERRUPTS (sonar) pin
#define TRIG 13
#define ECHO 2
#define THROTTLE_RX 5 //throttle from RX
#define SWITCH_RX 6   //switch from RX
#define THROTTLE 11   //throttle to auto-pilot
#define INTERRUPT 0

// acqusition time intervals:
// Sonar_Bw = 20 Hz
unsigned long sonarTimeInterval = 50000;
unsigned long trigTimeInterval = 13;

// variables related to the switch function
byte state=0;
//identifies the state of the square
//wave read on THROTTLE_RX and SWITCH_RX
byte stateSwitch=0;
//identifies the state of the square
//wave read on SWITCH_RX during
//the switchStick-HIGH cycle
int flag = 0;
//identifies which pin the square wave
//comes from: 0 - throttle, 1 - switch
boolean enableBypass = false;
//specifies when arduino
//must come back to switchStick-LOW cycle
unsigned int time = 0;
//during the switchStick-LOW cycle, stores the
//duty cycle of the SWITCH_RX, using TimerOne
unsigned int timeSwitch = 0;
//during the switchStick-HIGH cycle, stores
// the duty cycle of the SWITCH_RX,
// using micros()
boolean switchStick = false;
//identifies the state of the switch stick on
//TX: LOW (1900 us) and HIGH (1100 us)

// all variables are in microseconds
unsigned long startEchoTime=0;
unsigned long startSonarTime = 0;
unsigned long startTrigTime = 0;
unsigned long startSwitchSquare = 0;
unsigned long echoLength = 0;
unsigned long currentTime = 0;

// state of sonar interrupt handlers
int interruptState = 0;
int oldInterruptState=0;
int trig = 0;
int oldTrig = 0;

// enable TRIG pin to be raised or lowered
int trigEnable = 1;
int firstChangingEdgeEvent = 0;

// PID-related variables
boolean firstTimeReadingSonar = true;
int initialConditionOnIntegrator = 0;
double proportional = 0;
double integrative = 0;
double derivative = 0;
double derivative_2 = 0;
double derivative_3 = 0;
double P = 0.29672;
double I = 0.01570;
double D = 0.10581;
double N = 3.59690;
double Ts = 0.05;
double r = 0; //setpoint
double y = 100; //measured output (sonar)
double e = 0; //error between r and y
double u = 0; //(throttle)
int microseconds = 0;
int uSoffset = 43;

// sonar interrupt handler
void changeEdge() {
  if(firstChangingEdgeEvent==0)
    // needed by the first cycle,
    // otherwise it mess up everything
    firstChangingEdgeEvent=1;
  else
    interruptState = 1-interruptState;
}

Servo throttle;

void setMicroseconds(Servo &s, int uS) {
  if (s.readMicroseconds() != uS) {
    s.writeMicroseconds(uS);
  }
}

void PID(int sonar) {
  if(sonar<500)
    y = sonar;

  if(firstTimeReadingSonar) {
    //setting initial conditions on the PID
    firstTimeReadingSonar = false;
    r = y;
    integrative = initialConditionOnIntegrator;
    derivative = 0;
    derivative_2 = r;
    derivative_3 = r;
  }
  e = r-y;

  // P action
  proportional = P*e;

  // I action
  integrative = integrative + I*Ts*e;

  // D action (second-order derivative)
  derivative = D/(D+N*Ts*P)*derivative -
            D*N*P/(D+N*P*Ts)*(y-derivative_2);
  derivative_2 = derivative_3;
  derivative_3 = y;

  // output command
  u = proportional+integrative+derivative;
  microseconds = (int)(u+0.5);

  //applying saturation
  if(microseconds>1900)
    microseconds=1900;
  if(microseconds<1100)
    microseconds=1100;
}

void setup() {
  // pin initialization
  pinMode(THROTTLE_RX,INPUT);
  pinMode(SWITCH_RX,INPUT);
  pinMode(THROTTLE,OUTPUT);
  pinMode(TRIG,OUTPUT);
  digitalWrite(THROTTLE_RX,HIGH);
  digitalWrite(SWITCH_RX,HIGH);

  // square wave reading initialization
  Timer1.initialize(2200);
  Timer1.restart();
  PCintPort::attachInterrupt(THROTTLE_RX,
                                   rise,RISING);
  // attach a PinChange Interrupt to the
  // THROTTLE_RX pin
}

void loop() {
  if(!switchStick) {

//** routine during the switchStick-LOW cycle

    switch (state) {
    //works with the square waves on
    //THROTTLE_RX and SWITCH_RX pins
    case RISING: //we just saw a rising edge
     if(flag==0) {
      PCintPort::detachInterrupt(THROTTLE_RX);
      PCintPort::attachInterrupt(THROTTLE_RX,
                                    fall,FALLING);
     }
     else {
       PCintPort::detachInterrupt(SWITCH_RX);
       PCintPort::attachInterrupt(SWITCH_RX,
                                    fall,FALLING);
     }
     state=255;
     break;
    case FALLING: //we just saw a falling edge
     if(flag==0) {
        PCintPort::detachInterrupt(THROTTLE_RX);
        initialConditionOnIntegrator = time;
        PCintPort::attachInterrupt(SWITCH_RX,
                                   rise,RISING);
       }
       else {
         PCintPort::detachInterrupt(SWITCH_RX);
         PCintPort::attachInterrupt(THROTTLE_RX,
                                    rise,RISING);
         /* disable throttle bypass when the **
         ** switch stick is rised ************/
         if(time<1500) {
           // read the switch stick state
           switchStick = true;
           // disable the interrupts
           PCintPort::detachInterrupt(
                       THROTTLE_RX);
           PCintPort::detachInterrupt(
                       SWITCH_RX);
           attachInterrupt(INTERRUPT,
                          changeEdge,CHANGE);
           PCintPort::attachInterrupt(
                  SWITCH_RX,riseSwitch,RISING);
           throttle.attach(THROTTLE);
           setMicroseconds(throttle,
                   initialConditionOnIntegrator+
                   uSoffset);
           startSonarTime = micros();
           // set some sonar-related variables
         }
         /********************************/

        }
        flag = 1-flag;
        state=255;
        break;
      }
    }


    else {

/*** routine during the switchStick-HIGH cycle
*****************************************/

      switch (stateSwitch) {
        //works with the square
        //waves on SWITCH_RX pin
      case RISING:
        //we just saw a rising edge
        PCintPort::detachInterrupt(SWITCH_RX);
        PCintPort::attachInterrupt(SWITCH_RX,
                         fallSwitch,FALLING);
        startSwitchSquare = micros();
        stateSwitch=255;
        break;
      case FALLING: //we just saw a falling edge
        PCintPort::detachInterrupt(SWITCH_RX);
        stateSwitch=255;
        time = micros()-startSwitchSquare;
        if(time>1500)
          enableBypass=true;
        else
          PCintPort::attachInterrupt(SWITCH_RX,
                          riseSwitch,RISING);
        break;
      }

      /******* enable throttle bypass when the
      *********switch stick is lowered *******/
```

```
      if(enableBypass) {
        enableBypass = false;
        switchStick = false;
        firstTimeReadingSonar = true;
        throttle.detach(); // detach the servo
        Timer1.initialize(2200);//restart Timer1
        Timer1.restart();
        state=255;
        PCintPort::attachInterrupt(
            THROTTLE_RX, rise,RISING);
            // re-attach the interrupts
        PCintPort::detachInterrupt(SWITCH_RX);
        detachInterrupt(INTERRUPT);

      }
      else {

        /************* reading the sonar, ******
        /************* acting on the throttle */

        // takes currentTime in microseconds
        currentTime = micros();

        // sonar every sonarTimeInterval
        if(trigEnable==1 &&
           currentTime-startSonarTime>=
           sonarTimeInterval) {
          trig = 1;
          trigEnable = 0;

          // zeroing time
          startSonarTime = currentTime;
        }

        //trig handler raising and lowering TRIG
        if(oldTrig!=trig) {
          oldTrig=trig;
          if(trig==1) {
            digitalWrite(TRIG,HIGH);
            startTrigTime = currentTime;
          }
          else {
            digitalWrite(TRIG,LOW);
          }
        }

        // interrupt handler computing and
        // sending echoLength
        if(oldInterruptState!=interruptState) {
          oldInterruptState=interruptState;
          if(interruptState==1) {
            startEchoTime = micros();
          }
          else {
            echoLength = (micros()-
                       startEchoTime)/58;
            trigEnable = 1;
            PID(echoLength);
            setMicroseconds(throttle,
                      microseconds+uSoffset);

          }
        }

        // let the TRIG pin high only during
                                trigTimeInterval
        if(trig==1&&currentTime-startTrigTime>=
                              trigTimeInterval) {
          trig = 0;
        }
        /*************************************
        ***END*********************************/

      }
    }
  }

  void rise()
  {
    state=RISING;
    Timer1.restart();
    Timer1.start();
    if(flag==0&&!switchStick)
        //writes the throttle pin to the
        //auto-pilot iff the switchStick is low
      digitalWrite(THROTTLE,HIGH);
        //and we read throttle square wave
  }

  void fall()
  {
    state=FALLING;
    time=Timer1.read();
    if(flag==0&&!switchStick)
      digitalWrite(THROTTLE,LOW);
  }

  void riseSwitch()
  {
    stateSwitch=RISING;
  }

  void fallSwitch()
  {
    stateSwitch=FALLING;
  }
```

Appendix C: Python scripts

- **flyduino_aquisition.py**

```
# Acquisition from the arduino sketch running
# on the flyduino
#
# by Paolo Forni, Flavio Tonelli
# Feb 20, 2012

import sys
import serial
import time
import string

def main():
  if(len(sys.argv)<4):
    print "Usage:␣python␣acquisition.py
␣␣␣␣␣␣␣␣␣␣<serialInterface>␣<acquisitionTime>
␣␣␣␣␣␣␣␣␣␣<throttleFile>␣<sonarFile>"
    sys.exit(2);

  throttleFile = open(sys.argv[3],"w")
  sonarFile = open(sys.argv[4],"w")

  acquisitionTime = int(sys.argv[2])
  arduino = serial.Serial(sys.argv[1],9600)
  timeStart = time.time()
  acquisitionState=0;

  while(time.time()-timeStart<acquisitionTime):
    rawString = arduino.readline()
    if(rawString[0:3]=="###"):
      if(acquisitionState==0):
        acquisitionState=1
      sonarFile.write(rawString[3:])
      continue
    if(acquisitionState==1):
      throttleFile.write(rawString);

if __name__ == "__main__":
    main()
```

Appendix D: Matlab® & Simulink models

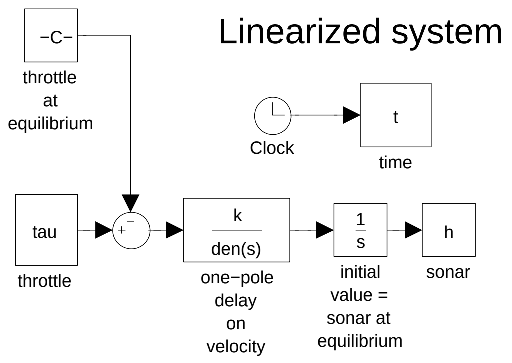

FIG. 8: Simulink model of the process

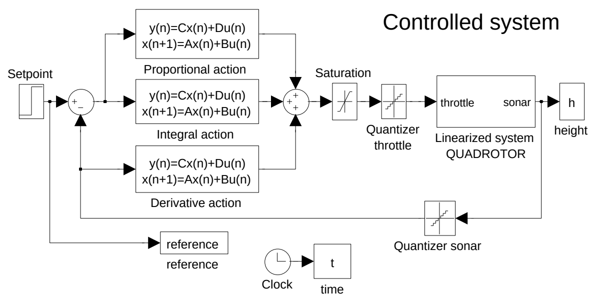

FIG. 9: Simulink model of the controlled system