

NATIONAL UNIVERSITY OF HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FALCUTY OF TRANSPORTATION ENGINEERING
DEPARMENT OF AEROSPACE ENGINEERING

-----o0o-----



Report #3

Implementation the Navigation and Control for Quadricopter

Student: Nguyễn Anh Quang

Student ID: V1002583

Supervisor: Prof Emmanuel GROLLEAU

Poitier, Spring 2015

Abstract

This report is the third report for the Project called” Implementation a flight command for a Quadricopter”.

This report focuses on the following details:

- *Using Pulse Width Modulation (PWM) to control a DC motor*
- *Navigation and motor control in the original ArduPilot framework*
- *Software in the Loop simulation and the required modifications to establish the simulation for this project*
- *Procedure to integrate new control system into ArduPilot framework*
- *New PID controller and the differences with the original one of ArduPilot*
- *Results of new PID controller in SITL simulation*

Mục lục - Table of Contents

Abstract.....	ii
Mục lục - Table of Contents	iii
Danh mục hình ảnh - List of Figures.....	iv
Chương 1	1
Navigation and Control in ArduPilot	1
1.1 Navigation and Control functions.....	1
1.1.1 Calculating Pulse Width Modulation (PWM) output for motors.....	1
1.1.2 Tasks for motors control and navigation in ArduCopter.....	3
1.1.3 Navigation system in auto mode.....	5
1.2 Utilisation of PID control in ArduPilot	8
Chương 2	10
Simulation and testing using Software in the Loop method (SITL)	10
2.1 An introduction about Software in the Loop simulation.....	10
2.2 SITL in ArduPilot	13
Chương 3	18
Contributions.....	18
3.1 Modifications in the original simulation code.....	18
3.2 Control signal outputs	22
3.3 New PID control method.....	24
3.4 Integrating new PID control into ArduPilot framework	26
3.5 New PID controller results	29
References.....	38

Danh mục hình ảnh - List of Figures

Figure 1-1: Example about using PWM to control output voltage	2
Figure 1-2: fast_loop in ArduPilot.....	3
Figure 1-3: Fast_loop tasks and their function.....	4
Figure 1-4: Earth frame navigation of ArduPilot	6
Figure 1-5: Body frame coordinate with AHRS_ORIENTATION = 0	6
Figure 1-6: Body frame coordinate with AHRS_ORIENTATION = 1	7
Figure 1-7: Navigation points in ArduPilot	8
Figure 1-8: PID gains of ArduPilot.....	8
Figure 1-9: ArduPilot auto mode calculation from inputs to output	9
Figure 2-1: An overview about Hardware in the Loop simulation system [3]	11
Figure 2-2: SITL architecture in ArduPilot.....	13
Figure 2-3: File structure of SITL simulation in ArduPilot.....	14
Figure 2-4: Command to start the SITL simulation	15
Figure 2-5: Some files and folders in autotest directory.....	16
Figure 2-6: Parameters in copter_params.parm	16
Figure 3-1: Some tasks for autopilot SITL simulation	19
Figure 3-2: Tasks are called alternately in the fly_ArduCopter command.....	20

Figure 3-3: Modifications in the tasks	21
Figure 3-4: New Test_mode to make the Quadricopter follows a specific flight plan.....	21
Figure 3-5: Some settings related to the built target in arducopter.py	22
Figure 3-6: Connecting ESC with motor and control board	22
Figure 3-7: Convert from angular speed to PWM output signal.....	23
Figure 3-8: Overshoot happens with the new PID controller	26
Figure 3-9: Create new directory in libraries and add it into ArduCopter.pde	27
Figure 3-10: New setting for motors_output task.....	28
Figure 3-11: Replacing old command with new one in auto_mode.....	28
Figure 3-12: Old command (left) and new one (right) with the tasks they will carry on.....	29
Figure 3-13: First flight plan	30
Figure 3-14: Second flight plan	30
Figure 3-15: A SITL simulation with new PID controller.....	31
Figure 3-16: A SITL simulation with new PID controller (2)	32
Figure 3-17: Overshoot and re-stabilize with new PID controller	32
Figure 3-18: Overshoot and re-stabilize with new PID controller (2).....	33
Figure 3-19: Difference between desired pitch and pitch result with new PID controller.....	34
Figure 3-20: Difference between desired roll and roll result with new PID controller.....	34
Figure 3-21: Difference between desired yaw and yaw result with new PID controller.....	35

Figure 3-22: Difference between desired Yaw and Yaw result with original PID controller	35
Figure 3-23: Difference between desired Roll and Roll result with original PID controller	36

Chương 1

Navigation and Control in ArduPilot

This part of the report presents an overview about ArduPilot Navigation and Control method, how it navigates and maintains control in the auto mode. In addition, this chapter will introduce the using of PID in ArduPilot and how to generate the PWM out of the result of the control method generate by Castillo in her previous work.

1.1 Navigation and Control functions

1.1.1 Using Pulse Width Modulation (PWM) to control DC motors.

Like any other electric motors controlling systems, ArduPilot uses the Pulse Width Modulation (PWM) as the control signal for the electric motors. Considering a square signal can be either zero voltage or twelve volts as shown in **figure 1-1**. This signal is cyclic and has the period of one minute. In the above figure, the twelve volts period lasts for thirty seconds, in this case, the output voltage of this signal is

$$U_{out} = \frac{30 \times 12 + 30 \times 0}{60} = 6V \quad (1)$$

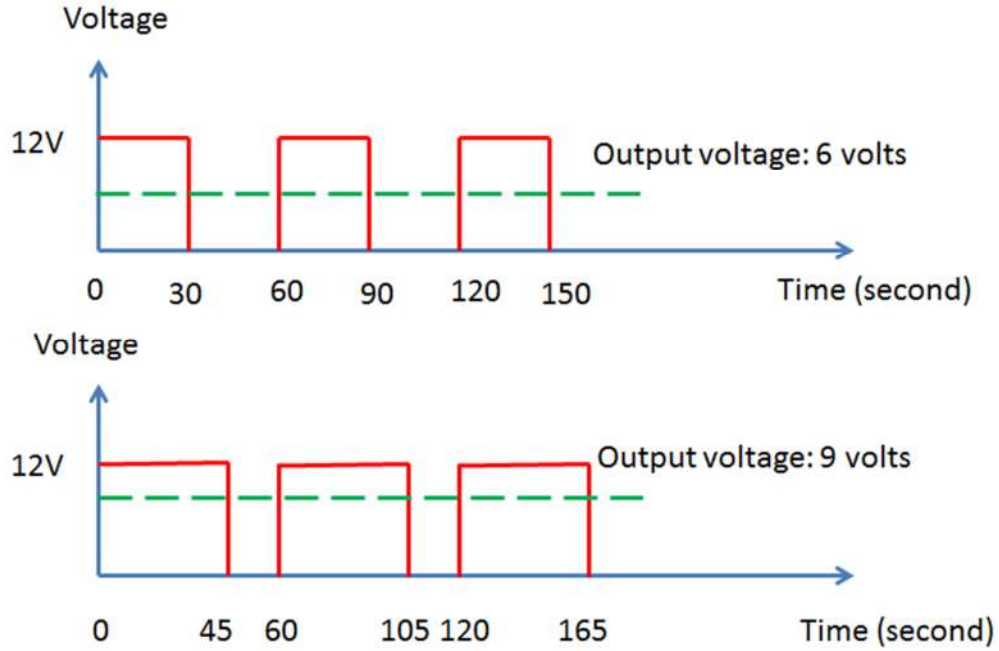


Figure 1-1: Example about using PWM to control output voltage

In the lower figure, the twelve volts output lasts for 45 seconds each period, as a result, the output voltage of the signal is nine volts. In case of a control system, since the frequency of the signal is big enough (which makes the duration of each period is really short), the output voltage can be considered as a constant signal with the average voltage above. If the programmers know the frequency of the signal, the **PWM value would be the duration whereas the signal is greater than zero** (the width of the positive pulse), and with this PWM, programmer can easily control the input voltage for the electric device. In general, the equation used to described the output value would be

$$U_{out} = \frac{t \times PositiveVoltage + (T - t) \times NegativeVoltage}{T} \quad (2)$$

with t is the period of the pulse (width of the positive pulse)

T is the period of the signal (width of the signal)

In order to make the motors understand the output PWM of the control board, an **Electric Speed Controller** (ESC) has to be installed for each motor. This device

will act as an on/off switch, translate the signal from the output pin into real electric signal and power the motor.

1.1.2 Tasks for motors control and navigation in ArduCopter

In ArduPilot, in particular ArduCopter, the task called **motors_output()** is in charge of calculating PWM output and transmitting this signal to ESC. Since controlling the motors is one of the most important tasks, this task is put into the **fast_loop**, which will be functioned at the rate of 400Hz for **FlyMaple**.

```
// Main loop - 400hz
static void fast_loop()
{
    // IMU DCM Algorithm
    // -----
    read_AHRS();

    // run low level rate controllers that only require IMU data
    attitude_control.rate_controller_run();

    #if FRAME_CONFIG == HELI_FRAME
        update_heli_control_dynamics();
    #endif //HELI_FRAME

    // send outputs to the motors library
    motors_output();

    // Inertial Nav
    // -----
    read_inertia();

    // run the attitude controllers
    update_flight_mode();
}
```

Figure 1-2: fast_loop in ArduPilot

The procedure of the tasks in the fast_loop as well as their functions can be described as in **figure 1-3**.

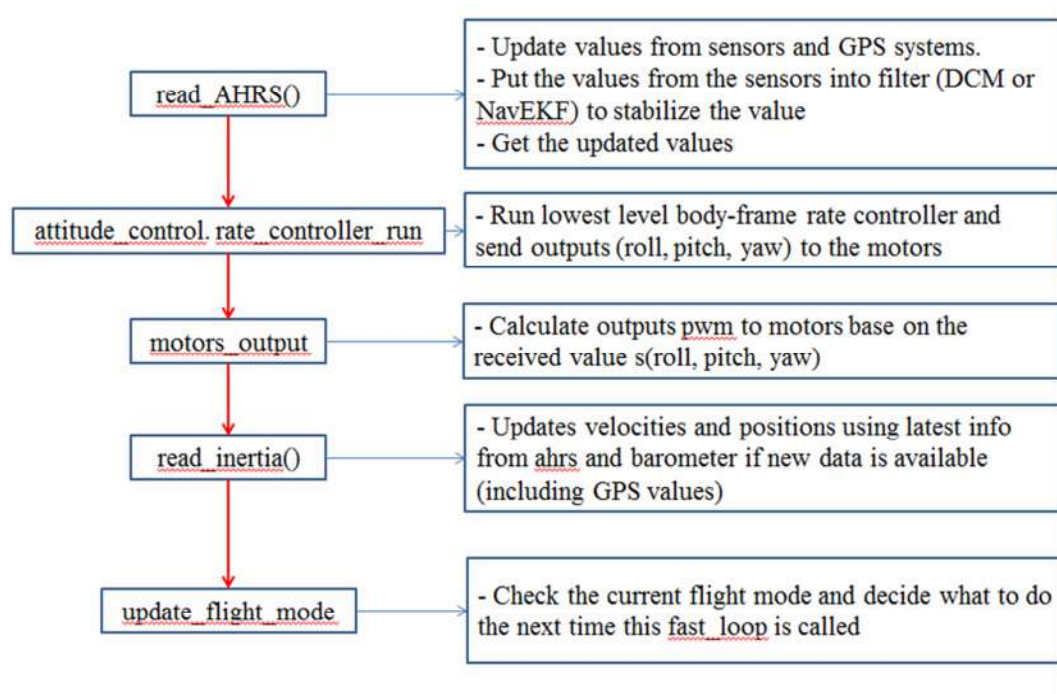


Figure 1-3: Fast_loop tasks and their function

From the figure above, there are some important notes, which are:

- Because ArduPilot supports not only the UAVs using Radio Control (RC) but also those flying follow a flight plan (auto mode), the most important functions in this loop are **read_AHRS**, **read_inertia** and **update_flight_mode**. In RC mode, these functions will act as the supporting systems to help the controller easily maintain the stability of the drone. In auto mode, these functions will do every in their own, directly control the UAV follows a planned mission.
- In fact, there are several tasks with different timer callback for sensors, gps systems... as mentioned in the scheduling part of report #1 [1]. In other words, the acquired values from sensors or GPS system are not memorized at the same time. For instance, the control system receives the GPS messages with the frequency of 50Hz, barometer is accumulated also at 50Hz, however, compass is read only at the rate of 10Hz. The difference in calling time here affect directly the autopilot systems since the values used in calculation do not have the same gained time. In order

to overcome this problem, ArduPilot uses the estimation and tries to predict the future values based on the past and the current values. These estimated values are then corrected every time the control system receives new ones. The estimation here is understandable; however, it is complicated and should not be modified since it will affect the hold autopilot system of this framework.

- ArduPilot provides users with several flight modes. More details about these modes can be found in [2]. As a result, the calculation for PWM outputs as well as roll, pitch, yaw rates of each flight mode is different. By modifying this calculation, programmers can generate their own control method and put it into ArduPilot framework to see the result.

1.1.3 Navigation system in auto mode

ArduPilot uses two different frames to navigate and calculate the required values in auto mode.

The first coordinate, which is called earth frame in ArduPilot, is in fact the inertial frame. *“It is an earth fixed coordinate system with origin at a define location. The unit vector i^i is directed the North, the j^i is to the East and the k^i is directed upward forming a clockwise trihedral with the others”* [3]. In ArduPilot, this frame is used mostly for navigation since it can determine easily the location of a point as long as we know the origin. More importantly, this frame is a fixed coordinate system, which means it will not change despite of the attitude of the drone. **Figure 1-4** gives an example about this coordinate and how it is used. As default, ArduPilot uses the **Home** location as origin, therefore, **waypoint 1** in this figure will have the location as X is zero and Y is a positive number. These values will not change while the UAV is moving unless the origin is changed.



Figure 1-4: Earth frame navigation of ArduPilot

The second coordinate is the body frame. This coordinate is fixed with the orientation of the control board, as can be seen in **figure 1-5** and **1-6**.

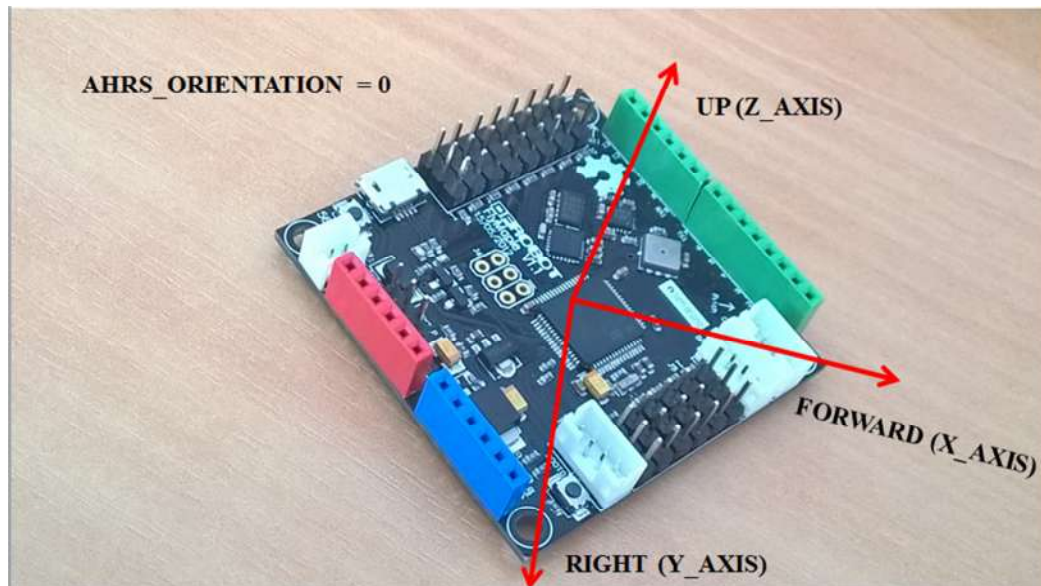


Figure 1-5: Body frame coordinate with AHRS_ORIENTATION = 0

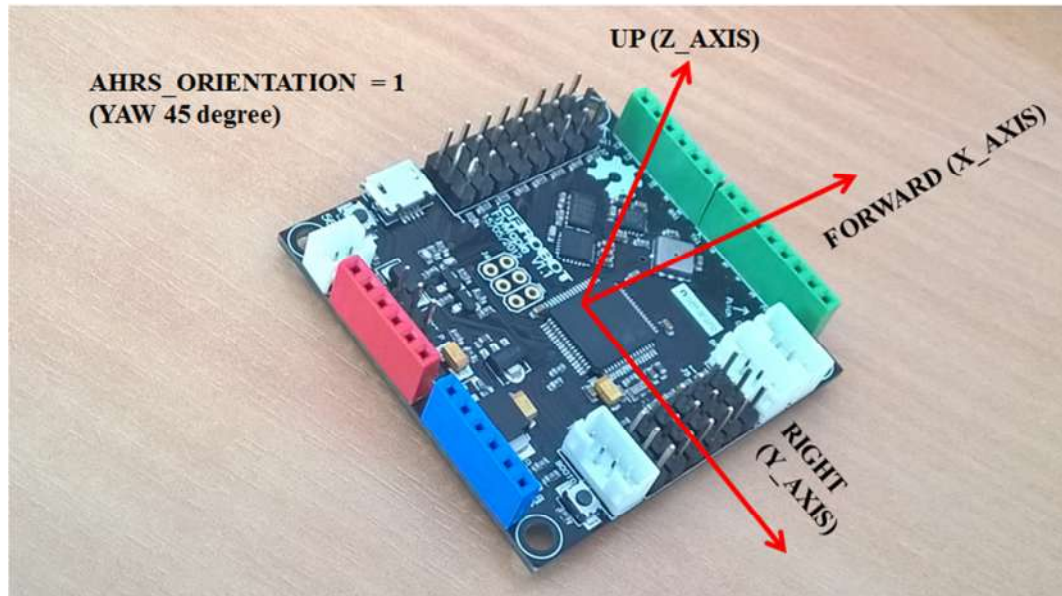


Figure 1-6: Body frame coordinate with AHRS_ORIENTATION = 1

ArduPilot uses this coordinate to calculate the outputs for autopilot. More details about them will be shown in the next part of this report.

Figure 1-7 gives a closer look in the navigation and autopilot of ArduPilot. There are five separated points used in ArduPilot auto mode:

- **Home:** As default, ArduPilot will take the launch point as Home, however, users can use GPS systems to modify this Home location.
- **Origin:** The previous waypoint which the UAV has just passed. In **figure 1-7**, Home (waypoint 0) is the Origin of the quadricopter
- **Destination:** The next waypoint in the mission. In **figure 1-7**, **waypoint 1** is the destination for the UAV.
- **Current point:** The location of the UAV at the time the system acquired. This location can be the exact location provided by GPS system, or it could be the estimated point as mentioned above.
- **Position target:** In order to make sure that the UAV will follow the desired path of the mission as close as possible, the path connecting Origin and Destination is divided into small segments by the points called position targets. By comparing the differences between current point and position targets in earth frame coordinate, the autopilot will generate the

desired values for roll, pitch, yaw, which will be then calculated to create the outputs for the motors.

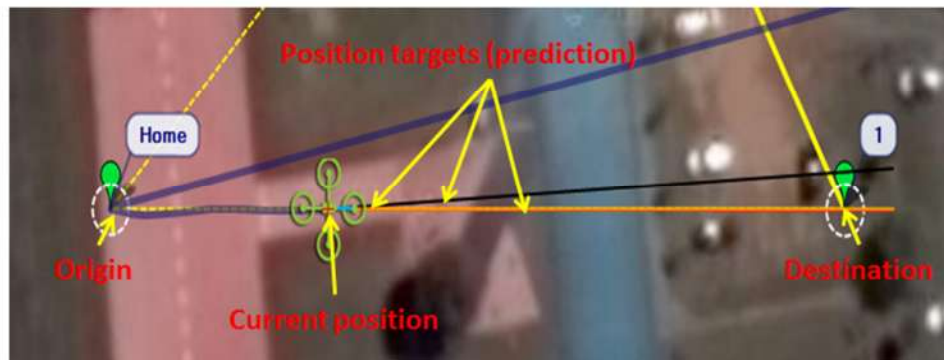


Figure 1-7: Navigation points in ArduPilot

1.2 Utilisation of PID control in ArduPilot.

By default, ArduPilot has five different PID loops to calculate the motors control signals in auto mode or to stabilize these signals in other modes. These PID control systems are separate with the others and each of them has their own gains as can be seen in **figure 1-8**. Among them, the PID controls related with Roll, Pitch, Yaw and Throttle will take account in every calculation. The other will only take place in Loiter mode.



Figure 1-8: PID gains of ArduPilot

Another noticed point in **figure 1-8** is that the PID gains for roll, pitch, yaw are called **Rated Roll** (P, I and D), **Rated Pitch** (P, I, and D) and **Rate Yaw** (P, I and D) respectively. This is also the idea about the PID controls of ArduPilot, as it use PID to control the **change rate** of the Roll, Pitch, and Yaw. In detail, the calculation from the beginning to the motor signals in ArduPilot can be explained as in **figure 1-9**.

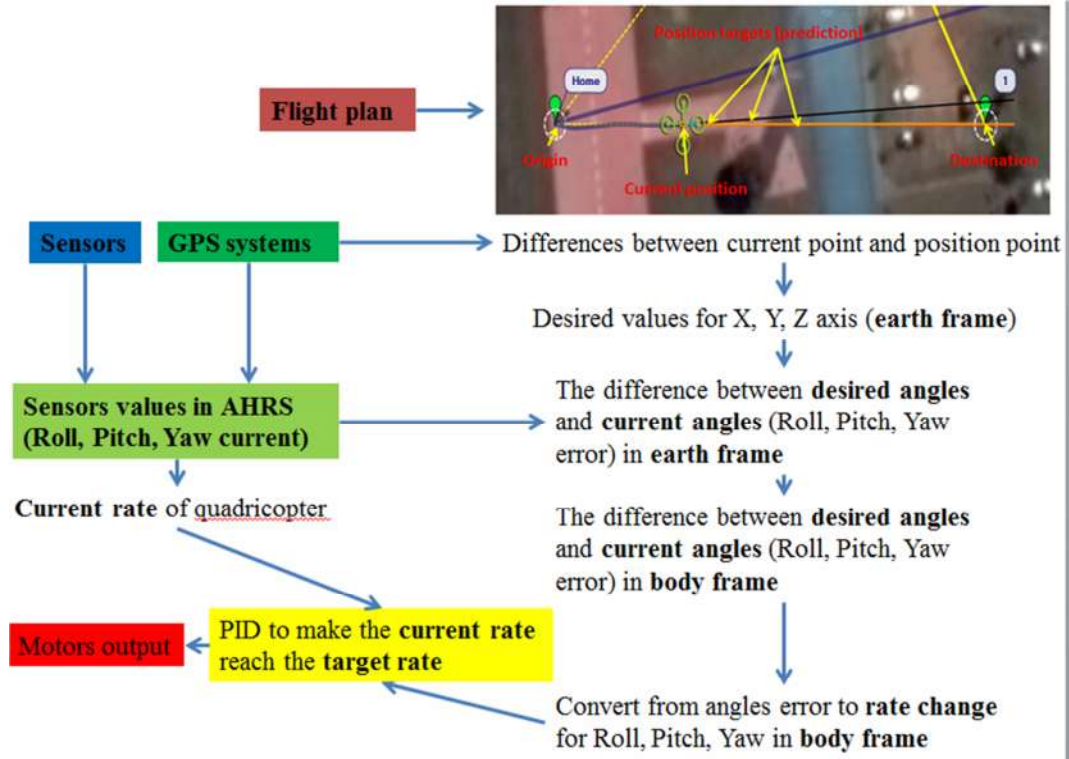


Figure 1-9: ArduPilot auto mode calculation from inputs to output

By using both coordinates, ArduPilot can use the benefits of the one to cover the other weakness. Earth frame (or inertial frame to distinguish with the real earth frame) gives the fixed position of each waypoint in relative with origin, meanwhile, body frame takes in account the orientation and rotation of the quadricopter, makes it easier to control. The control of rate of change, instead of roll, pitch, yaw directly also gives ArduPilot some advantages. These advantages will be discussed further in the integration of new PID control of this report.

Chương 2

Simulation and testing using Software in the Loop method (SITL)

This part introduces the using of Software In The Loop (SITL) to simulate and test the ArduPilot code before using it in a real models. This chapter also presents the required modifications with the simulation code to create an ideal environment for testing new code as well as control method.

2.1 An introduction to Software in the Loop simulation

Simulation is one of the most important steps for any design and programming procedure in general. Although it cannot describe and take in account every situation that could happen in the real world, testing a system in simulation will help developers not only understanding the behaviors of that system in the real world but also noticing what they could do to optimize that system. For an UAV, control code will need to be tested many times in virtual world to make sure that it could work properly and prevent any unfortunate error due to control error.

There are two simulation methods that are usually used, **Hardware in the Loop** (HITL) and **Software in the Loop** (SITL).

HITL simulation is a technique that is use the real hardware (an embedded system for example) with the simulated input values. By giving the control board the values that would be given to it by the real sensors in the real situation, developers could learn about the behavior of that system. The data from this simulation will help developers predict the react of the same hardware in the real world in case of facing a familiar problem and then modify the control code if necessary. In short, this simulation uses **real hardware** in a **virtual environment** and learn about **real behavior** that of that hardware.

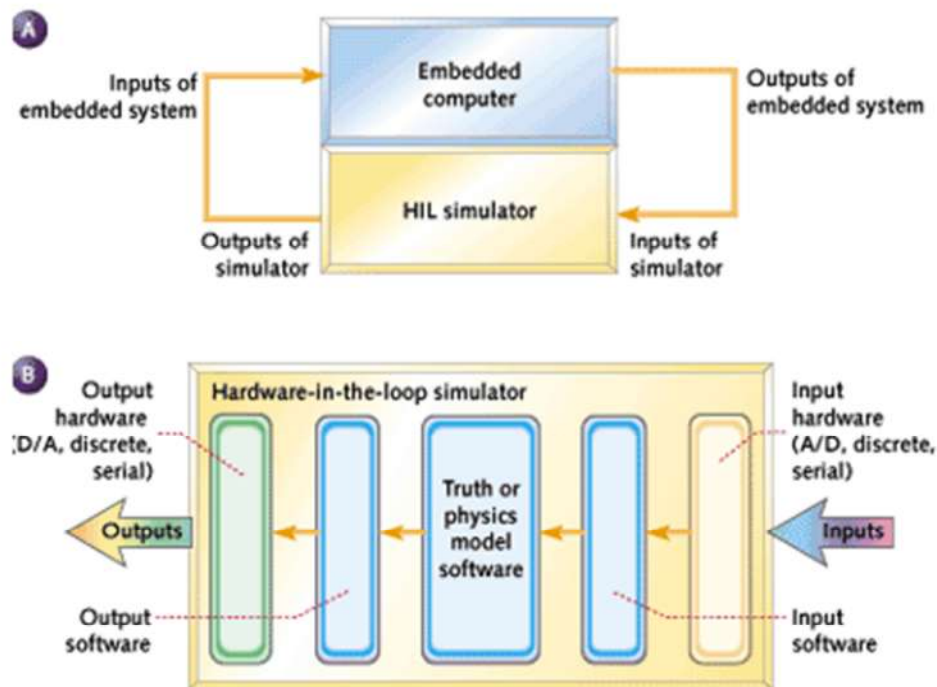


Figure 2-1: An overview about Hardware in the Loop simulation system [4]

The major difference between **SITL** and **HITL**, as can be seen from their name, is that **SITL** does not need a real hardware in the simulation process. Indeed, with **SITL**, the hardware is also simulated as well as the testing environment. Both **HITL** and **SITL** have their own benefits as well as drawbacks, as can be seen from **table 1**.

Table 1: Advantages and disadvantages of HITL and SITL

	HITL	SITL
Pros	<ul style="list-style-type: none">- Testing the real behavior of a real system in situations which cannot be done with real tests because of endanger.- The result is more reliable since it is the react of a real hardware.- More safety as well as saving more time and money comparing to real system test.- Can be done in early stage and with separated components as well as a whole system.	<ul style="list-style-type: none">- Do not need a real hardware.- Saving more time and money than HITL.- Can test many different situations, some of which cannot be done in HITL.- Easier to implement.
Cons	<ul style="list-style-type: none">- In some situations, creating the virtual inputs is not simple, need to understand clearly about the testing environment.- For some systems, the real hardware is not only complicated but also require a lot of space.	<ul style="list-style-type: none">- The result is not as reliable as HITL since it just the behavior of a virtual hardware.- Need time to create as well as validate the virtual hardware, comparing its behavior with the real one.

In fact, depending on the objectives of the test, developers will decide which simulation should be used. However, in case of ArduPilot, SITL seems to have more advantages because of two limitations of HITL:

- *It cannot run all of the autopilot code, as the low-level driver code will not see suitable inputs for a test flight when the hardware is sitting on your desk.*
- *You cannot use the sort of advanced programming tools (such as debuggers and memory checkers) that are so useful in normal C++ development. [5]*

Therefore, in ArduPilot, developers have created the necessary tools for users around the world testing their autopilot systems with Software in the Loop simulation.

2.2 SITL in ArduPilot

Figure 2-2 gives a simple description about SITL architecture in ArduPilot.

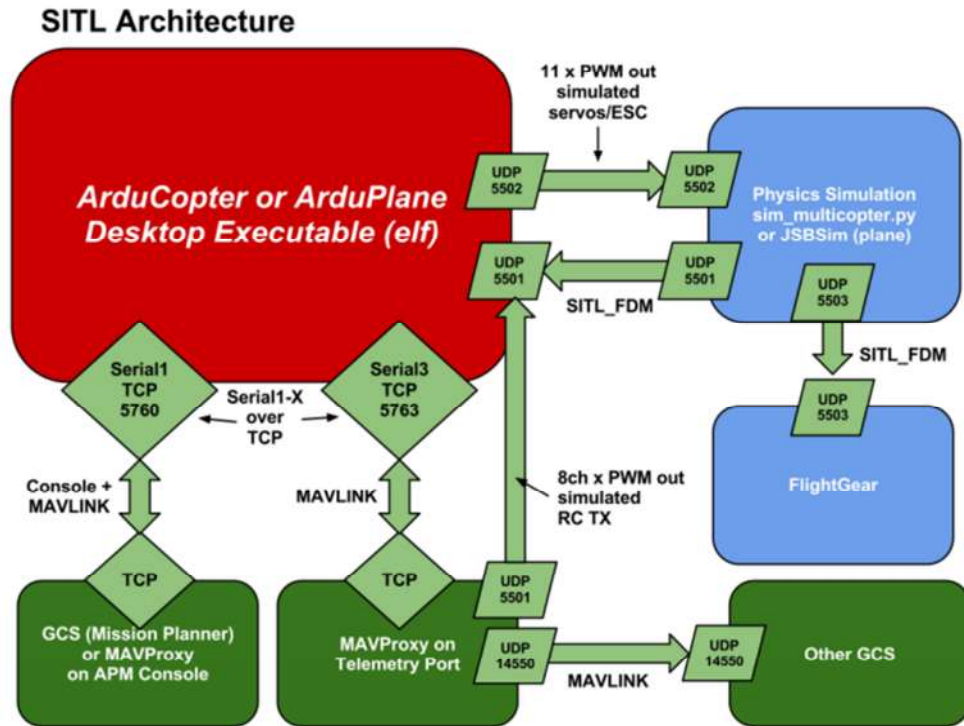


Figure 2-2: SITL architecture in ArduPilot

As can be seen, SITL architecture in ArduPilot includes three main parts. The first part is the *.elf file or the autopilot firmware built by ArduPilot framework. The architecture of ArduPilot as well as the process to create elf file can be found in report #1 [1]. The second part related to the Ground Control Stations and the communication between it and the autopilot system. More detail about this communication (MAVLink protocol, choosing ground control station...) can be seen in report #2 [6]. In this part, only the blue boxes are the one need to be concerned. These boxes represent the virtual hardware and the virtual environment for SITL simulation. By putting the built firmware into this virtual hardware, users can simulate the behavior of the hardware (in our case a quadricopter) and test the new controller.

The SITL in ArduPilot is constructed in Python, its files structure can be described as in **figure 2-3**. The files in this figure can be found in the **Tools/autotest** directory of ArduPilot

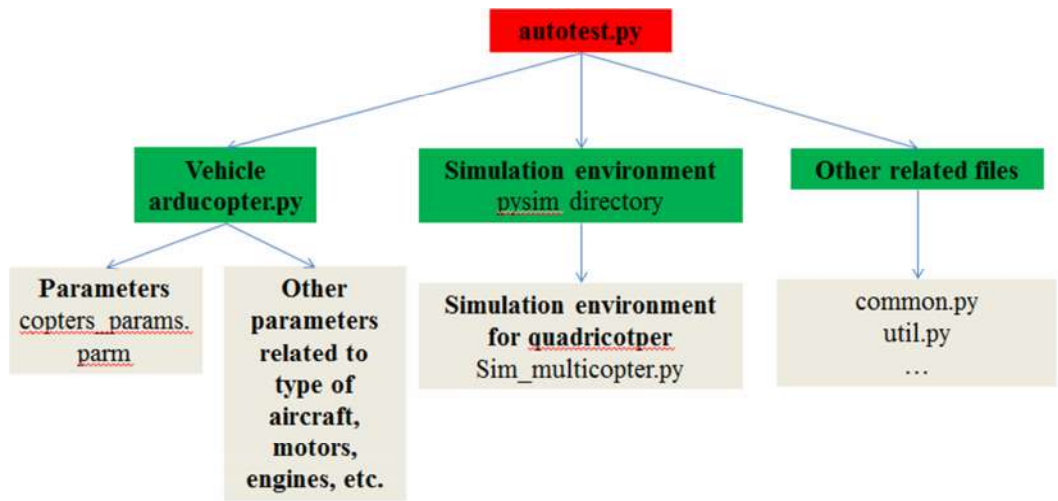


Figure 2-3: File structure of SITL simulation in ArduPilot

As can be seen from the figure above, **autotest.py** is the main file of the simulation, which defines not only the commands that could be used to build the systems but also the simulation process. For instances, some main commands that could be used for building in simulation including in **autotest.py** and their functions can be seen in **table 2**.

Table 2: Some command can be used with autotest.py

Command	Function	Located file
prerequisites	check we have the right directories and tools to run tests	autotest.py
build.All	run the build_all.sh script	autotest.py
build2560.ArduPlane	build the firmware for board Mega2560 specific for ArduPlane	util.py
build.ArduPlane	build the firmware for ArduPlane with target board is defined as sitl	util.py
defaults.ArduPlane	get default parameters for ArduPlane	autotest.py
fly.ArduPlane	use the firmware has just been built in a SITL simulation test	arduplane.py
build2560.APMrover2	build the firmware for board Mega2560 specific for APMrover	util.py
build.APMrover2	build the firmware for APMrover	util.py

	with target board is defined as sitl	
drive.APMrover2	use the firmware has just been built in a SITL simulation test	apmrover2.py
build2560.ArduCopter	build the firmware for board Mega2560 specific for ArduCopter	util.py
build.ArduCopter	build the firmware for ArduCopter with target board is defined as sitl	util.py
fly.ArduCopter	use the firmware has just been built in a SITL simulation test	arducopter.py

In this project, our board is FlyMaple and our target vehicle is a quadricopter; therefore, the command in **figure 2-4** could be used to start the simulation process.

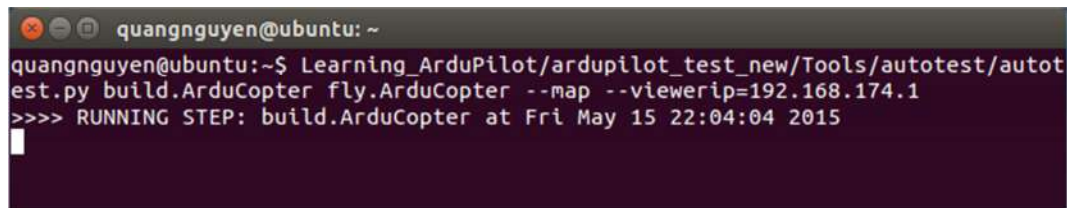
A terminal window with a dark background and light text. The prompt is 'quangnguyen@ubuntu: ~'. The command entered is 'Learning_ArduPilot/ardupilot_test_new/Tools/autotest/autotest.py build.ArduCopter fly.ArduCopter --map --viewerip=192.168.174.1'. The output shows '>>> RUNNING STEP: build.ArduCopter at Fri May 15 22:04:04 2015' followed by a cursor.

Figure 2-4: Command to start the SITL simulation

For each kind of simulation target, there are two different groups of parameter. The first group is the physical parameters of that target such as its weight, type of motor, type of propeller, etc. These values are packed in several directories such as **aircraft** or **param_metadata** and should not be modified unless users have certain knowledge about the structure of this simulation as well as about python. The second parameter group, however, is easier to modify. For ArduCopter, these values are put into **copter_params.parm** file. As can be seen in **figure 2-6**, the values in the second group related to the values that the hardware would get during the calibrating process of a real UAV. By changing these values, or adding some more, users could modify the virtual hardware in the SITL simulation as close as the real hardware to increase the reliability of the result.

Name	Date modified	Type	Size
aircraft	4/9/2015 8:27 PM	File folder	
apm_unit_tests	4/9/2015 8:27 PM	File folder	
ArduPlane-Missions	4/9/2015 8:27 PM	File folder	
jsbsim	4/9/2015 8:27 PM	File folder	
param_metadata	4/9/2015 8:27 PM	File folder	
pysim	5/2/2015 2:33 PM	File folder	
test_original	4/23/2015 2:57 PM	File folder	
web	4/9/2015 8:27 PM	File folder	
web-firmware	4/9/2015 8:27 PM	File folder	
ap1.txt	3/2/2015 3:56 PM	TXT File	1 KB
apmrover2.py	3/2/2015 3:56 PM	Python File	6 KB
apmrover2.pyc	4/21/2015 2:55 PM	Compiled Python ...	6 KB
arducopter.py	5/13/2015 7:48 PM	Python File	42 KB
arducopter.pyc	5/13/2015 7:48 PM	Compiled Python ...	28 KB
ArduPlane.parm	3/2/2015 3:55 PM	PARM File	2 KB
arduplane.py	3/2/2015 3:56 PM	Python File	17 KB
arduplane.pyc	4/21/2015 2:55 PM	Compiled Python ...	16 KB
autotest.py	3/2/2015 3:56 PM	Python File	15 KB

Figure 2-5: Some files and folders in autotest directory

1	FRAME	0
2	MAG_ENABLE	1
3	FS_THR_ENABLE	1
4	BATT_MONITOR	4
5	CH7_OPT	7
6	COMPASS_LEARN	0
7	COMPASS_OFS_X	5
8	COMPASS_OFS_Y	13
9	COMPASS_OFS_Z	-18
10	FENCE_RADIUS	150
11	RC1_MAX	2000.000000
12	RC1_MIN	1000.000000

Figure 2-6: Parameters in copter_params.parm

Besides the files mentioned above, there are some more supporting files such as the directory including the code to define the testing environment, the directory to export the result, etc. All of these files are also constructed in Python and also should not be modified.

The idea about the SITL simulation of ArduPilot is that even one developer with a simple personal computer could carry on the simulation and testing their new code. Therefore, in order to use the Software in the Loop simulation, a virtual machine using Ubuntu Operation System has to be set up. This virtual

machine will be the environment to build and simulate the hardware. The other computer, which is the host machine, will act as the ground control station and communicate with the virtual drone via UDP connection. More details about this setting up as well as other information related to using SITL simulation for ArduPilot can be found in [7] and [8].

Chương 3

Contributions

After understanding the basics of autopilot and navigation as well as the SITL architecture of ArduPilot, some modification has to be done before integrating the new control codes into ArduPilot. This chapter will include the following parts:

- *The modifications in the simulation code*
- *Solution for the motor control signals*
- *The new PID controller*
- *Procedure to integrate new control code into ArduPilot code base*
- *Result with new PID control code*

3.1 Modifications in the original simulation code

As mentioned above, all of the simulation code can be found in the tools/autotest directory. Although it requires many code lines to implement a simulation for ArduPilot, developers only need to change some lines in **arducopter.py**, **multicopter.py** and **parcopter_params.parm** to set a new simulation suitable for their target.

As default, the SITL simulation for a quadricopter or ArduCopter in general includes many specific tasks. **Figure 3-1** represents some of these tasks. After being defined, these tasks will then be called alternately in the main function called **fly_ArduCopter**, which will be called directly by **autotest.py**.

```
def arm_motors(mavproxy, mav):
    '''arm motors'''
    print("Arming motors")
    mavproxy.send('switch 6\n') # stabilize mode
    wait_mode(mav, 'STABILIZE')
    mavproxy.send('rc 3 1000\n')
    mavproxy.send('rc 4 2000\n')
    mavproxy.expect('APM: ARMING MOTORS')
    mavproxy.send('rc 4 1500\n')
    mav.motors_armed_wait()
    print("MOTORS ARMED OK")
    return True

def disarm_motors(mavproxy, mav):
    '''disarm motors'''
    print("Disarming motors")
    mavproxy.send('switch 6\n') # stabilize mode
    wait_mode(mav, 'STABILIZE')
    mavproxy.send('rc 3 1000\n')
    mavproxy.send('rc 4 1000\n')
    mavproxy.expect('APM: DISARMING MOTORS')
    mavproxy.send('rc 4 1500\n')
    mav.motors_disarmed_wait()
    print("MOTORS DISARMED OK")
    return True

def takeoff(mavproxy, mav, alt_min = 30, takeoff_throttle=1700):
    '''takeoff get to 30m altitude'''
    mavproxy.send('switch 6\n') # stabilize mode
    wait_mode(mav, 'STABILIZE')
    mavproxy.send('rc 3 %u\n' % takeoff_throttle)
```

Figure 3-1: Some tasks for autopilot SITL simulation

```

try:
    mav.wait_heartbeat()
    setup_rc(mavproxy)
    homeloc = mav.location()

    # Arm
    print("# Arm motors")
    if not arm_motors(mavproxy, mav):
        failed_test_msg = "arm_motors failed"
        print(failed_test_msg)
        failed = True

    print("# Takeoff")
    if not takeoff(mavproxy, mav, 10):
        failed_test_msg = "takeoff failed"
        print(failed_test_msg)
        failed = True

    # Fly a square in Stabilize mode
    print("#")
    print("##### Fly a square and save WPs with CH7 switch #####")
    print("#")
    if not fly_square(mavproxy, mav):
        failed_test_msg = "fly_square failed"
        print(failed_test_msg)
        failed = True

    print("# Land")
    if not land(mavproxy, mav):
        failed_test_msg = "land failed"
        print(failed_test_msg)
        failed = True

```

Figure 3-2: Tasks are called alternately in the `fly_ArduCopter` command

Although a good simulation test required many tasks to find the behavior of the system in different situations, some of these default tasks are not appropriate for this project. As can be seen in **figure 3-1**, as a part of Software in the Loop simulation, the Radio Control (RC) signals are also simulated. This is good for other simulation tests, however, in this project, RC signals will not be used to control the quadricopter in a real flight. Therefore, these tasks must be excluded from the main command **fly_ArduCopter**. To test the new control code, `fly_ArduCopter` only need two tasks, as represented in **figure 3-3**. The first task will arm the motors and the latter will make the quadricopter fly in auto mode follow a specific flight plan. Although the `arm_motors` command also used the simulated RC signal, this process can be override in a real quadricopter by a simple function pretending as it is sending the signal from the RC. This function

as well as the information about pre-setting of a real quadricopter will be mentioned further in the next report.

```
try:
    mav.wait_heartbeat()
    setup_rc(mavproxy)
    homeloc = mav.location()

    # Arm
    print("# Arm motors")
    if not arm_motors(mavproxy, mav):
        failed_test_msg = "arm_motors failed"
        print(failed_test_msg)
        failed = True

    print("# Takeoff")

    ##### Start flight test#####
    #Self written mode
    print("#Testing Ardupilot new code");
    if not Test_mode(mavproxy,mav):
        failed_test_msg = "Test_mode failed"
        print(failed_test_msg)
        failed = True
    print("End Test")
```

Figure 3-3: Modifications in the tasks

```
def Test_mode(mavproxy,mav):
    '''Self written code for autoflight test'''
    print('You should see this\n')
    print('Waypoint is loading\n')
    # Fly mission #1
    print("# Load copter_mission")
    if not load_mission_from_file(mavproxy, mav, os.path.join(testdir, "Pitch_Roll_Test5.txt")):
        print("load copter_mission failed")
        return False
```

Figure 3-4: New Test_mode to make the Quadricopter follows a specific flight plan

The details of **Test_mode** task is shown in figure 3-4. In these lines, if users need to change the flight plan for testing the control code in different cases, only the name of the mission file “*Pitch_Roll_Test5.txt*” need to be changed. After being built and implemented, the quadricopter will follow the new mission as defined in that txt file.

Beside the structure of the tasks, other configurations also have to be done if necessary to make the simulation as close as the real hardware. For example, in default, ArduCopter simulation target is a quadricopter plus frame. Depends on the real target, this frame can be changed by simply change the setting in the arducopter.py file as shown in figure 3-5. In addition, the location of home, the default heading... can also be change in the setting part of arducopter.py.

```
FRAME='+'  
TARGET='sittl'  
HOME=mavutil.location(-35.362938,149.165085,584,270)  
AVCHOME=mavutil.location(40.072842,-105.230575,1586,0)  
  
homeloc = None  
num_wp = 0
```

Figure 3-5: Some settings related to the built target in arducopter.py

3.2 Control signal outputs

As mentioned in the first chapter of this report, for UAV using electric motors, an ESC is installed to help the users to control the motor. **Figure 3-6** gives an easy example about connecting the control board, battery, ESC with the motor.

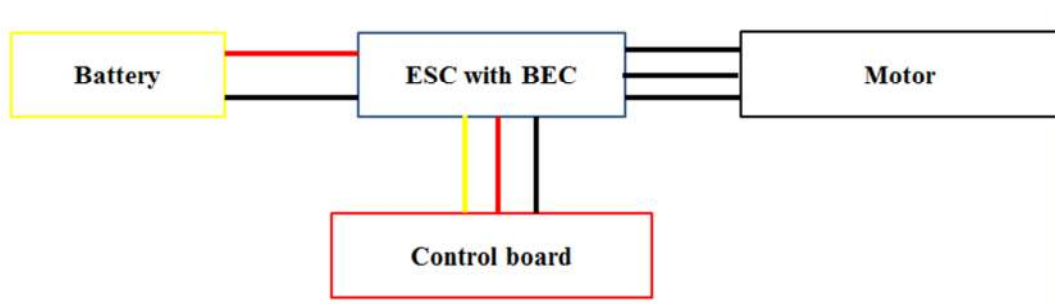


Figure 3-6: Connecting ESC with motor and control board

In the figure above, an ESC with a **b**attery **e**liminator **c**ircuit (BEC) is used to supply the power to the control board. From any ESC with a BEC, there are three wires with the color as shown in the figure. Among three of them, the yellow wire is the signal transmitting wire, the red line is the hot line and the black one is the ground line. By connecting three of them into an output pin of a

control board, the board will not only have the power to operate but also the connection to transmit the control signals to ESC to control the motor. From the previous work, as can be seen in [3], the final output of the calculation is the angular speed Ω (rad/s). However, the control signal for the motor must be PWM as mentioned in chapter one. The calculation process from rad/s to PWM for each motor can be implemented, however, it is not necessary. In additionally, this calculation depends on various characteristics of the real hardware such as the timer callback to send the signal to ESC, the output frequency of the ESC, etc. Since ESC can be calibrated to understand what is the minimum input as well as the maximum input it could receive, which will become the minimum throttle and maximum throttle respectively, a simple calculation procedure will be used, as shown in **figure 3-7**.

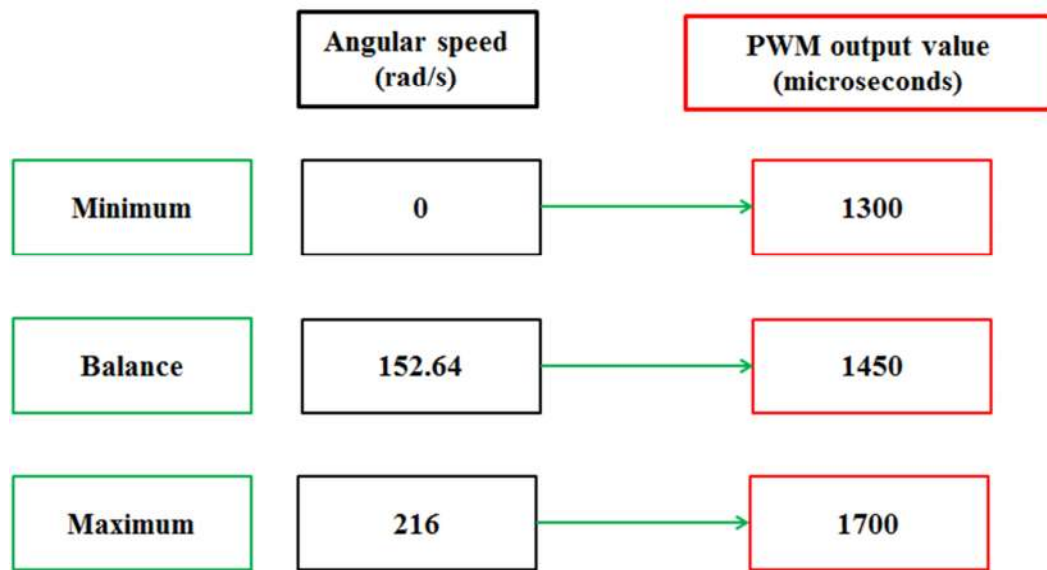


Figure 3-7: Convert from angular speed to PWM output signal

The constant value for the angular speed of the rotors is from the previous work, as shown in [3]. The PWM output values are from the real system and its behavior with different PWM values. In general, this value is between 1000 and 2000, depending on various things as mentioned above. With the hardware in the SITL simulation of ArduPilot, 1000 will be the minimum and will be the disarmed value; therefore, to prevent the auto disarm, a bigger value is chosen;

1450 will be the value at which the quadricopter will hold its altitude, the maximum value can reach up to 1800 – 1900. However, if the gap between the balance output and the maximum output is too great, the quadricopter will become easily unstable due to the sudden change every time the quadricopter reaches the target altitude. Any angular speed between the minimum and balance as well as the balance and maximum, its PWM will be calculated linearly. Although this calculation might be not one hundred percent correctly, due to the fact that the system will automatically adjust its angular speed by using control method such as PID, this calculation is considered as the easiest way to convert from angular speed to PWM output signal.

Nevertheless, more test and calculation must be done for the real quadricopter, in particularly determining the balance value for PWM control signal. The more precisely this value, the more stability the real system could achieve.

3.3 New PID control method

The PID controller described in this report is the result from the previous work. More details about this PID model as well as other control model can be found in [9] and [3]. From the Simulink model of Matlab, using Gene-auto¹, a C/C++ code structure has been created and ready to be integrated into ArduPilot.

Comparing between this new PID controller and the original one of ArduPilot, there is one major difference: the control target. The original controller controls the **change rate of Roll, Pitch, Yaw** as mentioned above. Meanwhile, the new

¹ *Gene-Auto is an open-source toolset for real-time embedded systems. The toolset takes as input a functional description of an application specified in a high-level modelling language (Simulink/Stateflow/Scicos) and produces C code as output. [10]*

PID controller **controls directly the Roll, Pitch, Yaw angle** of the quadricopter. There are some advantages along with the disadvantages for this changing.

The most important advantage of the original controller is that it prevents the overshoot better than the new one. In case of the error is too big, for example changing heading from North (0 yaw degree in earth frame) to South (180 yaw degree), new PID controller might have big overshoot. The process of both controllers for this example can be described as in **figure 3-8**. At the beginning, there is no yaw change rate since the quadricopter is stabilize and holding its attitude. The error of yaw rate and error of yaw angle is the first line. With this difference, PID controller will change the motor outputs and create the difference between motors to make the quadricopter rotates around its z-axis. In the second loop, because now the quadricopter has started to rotate, the error of yaw rate is now smaller than in the first loop, this difference will once again calculate with the PID, finally, the current yaw rate will reach the target yaw rate defined by the developer. The system yaw rate is now stabilized. Meanwhile, in case of using the PID to control directly the yaw, the error of yaw angle always meet its maximum value, which is set to prevent the system rotates too fast. This maximum error is read again and again, as a result, the PID controller will try to increase the rotate speed. Moreover, the saved error of the system will also increase. These phenomena will lead to two consequences. The first one is that now the rotate speed will increase continuously, when the heading is finally reach 180 degree, the rotation speed is now too fast, it will take more time to slow down, change the rotate direction and then come back to the desired heading. Secondly, since in PID controller, the error from the past always a part of the calculation for the current output value, and if the error becomes too big, while the heading finally reach South and then past it, the controller will not response fast enough for this change. Consequently, overshoot will happen as well as settling time will be lengthened.

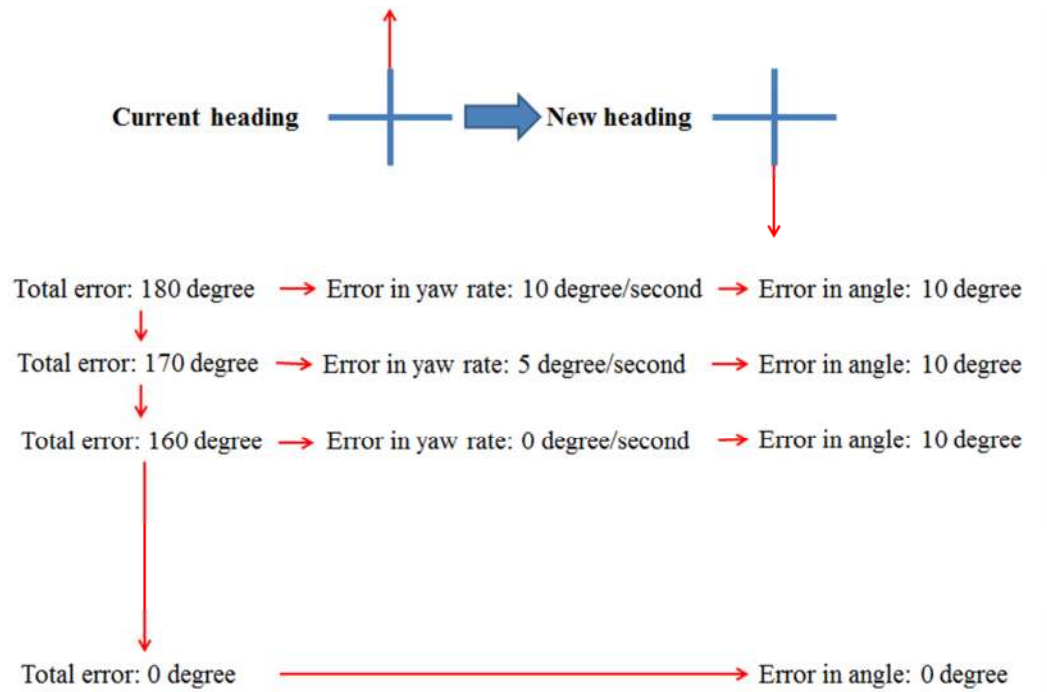


Figure 3-8: Overshoot happens with the new PID controller

The problem described above will also happen in case of descending or ascending, which requires not only the precise but also the fast response.

However, the new PID controller also has some advantages. Since this controller does not depend on the yaw rate, which is usually measured by an accelerometer; therefore, it will neglect unwanted system error of this sensor. In addition, the new PID controller will reduce the amount of calculation, which will save some time for the controller to handle other tasks. This benefit is essential for both old controller boards with slow calculation speed and new ones with high main loop frequency.

3.4 Integrating new PID control into ArduPilot framework



Because the code structure of ArduPilot has been developed so well, the most important objective when integrating new code into this framework is that it should not violate this structure. The general idea for this integration is that new code will work as it is the original one, with whole of the framework

characteristics. The procedure of integrating new PID controller into ArduPilot framework would be:

1. Create a new directory in the library containing the C/C++ files created by Gene-auto and then add the new library into the main code into ArduCopter.pde file as shown in **figure 3-9**. Check to find out if the integration is good by building the firmware by the cross-compiler.

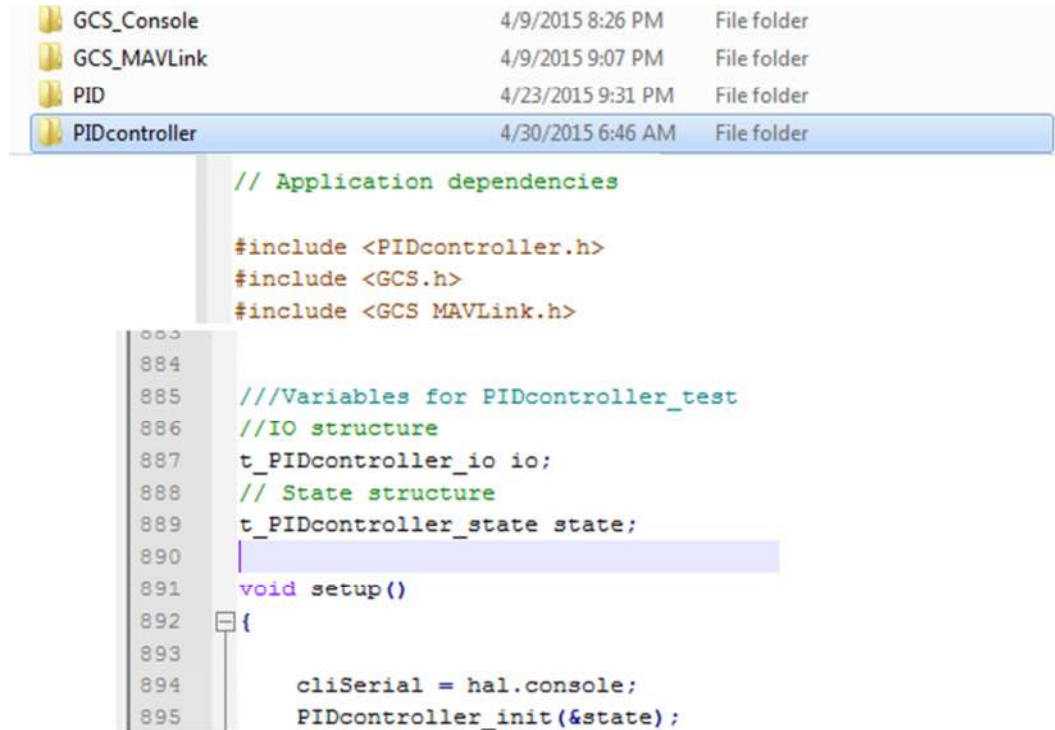


Figure 3-9: Create new directory in libraries and add it into ArduCopter.pde

2. Create new tasks to send new PWM outputs to the ESC. The structure of these tasks should base on the structure of **motors_output** in the **fast_loop** in ArduCopter.pde. Among the new tasks, there should be a task converting the angular speed into PWM output as mentioned above.
3. Change the setting of **motors_output** as shown in **figure 3-10** to help the system not confusing between new PWM code and old PWM code. For a real quadricopter, a new task to simulate the arming process will be generate, however it is not necessary for this simulation.

```

// send outputs to the motors library
if (!motors.armed()) {
    motors_output();
}

```

Figure 3-10: New setting for motors_output task

4. Create a new command in control_auto.pde. This command will replace the old one in case of auto waypoint navigation in auto mode. In fact, there are other solutions to integrate the new controller into ArduPilot framework. However, using the commands of the control_auto.pde is one of the simplest ways. This changing will not affect the structure of other control method, in particular controlling with the RC. This file also includes all of the necessary command for navigation and with some small modifications, our new controller will operate as it is the original code.

```

static void auto_run()
{
    // call the correct auto controller
    switch (auto_mode) {

        case Auto_TakeOff:
            auto_takeoff_run();
            break;

        case Auto_WP:
        case Auto_CircleMoveToEdge:
            //auto_wp_run(); ← Old command
            auto_wp_pid(); ← New command to navigate waypoint and using new PID controller
            break;
    }
}

```

Figure 3-11: Replacing old command with new one in auto_mode

5. Complete the code for the new command above. In order to use every facilities provided by ArduPilot framework (getting values from sensors, using the filters to stabilize the attitude values, estimating the position...), the code structure of this new command must bases on the original. Moreover, because this command will use the new PID controller, some additional lines related to this calculation will be also in put into this command. The idea of this new command can be found in the figure below.

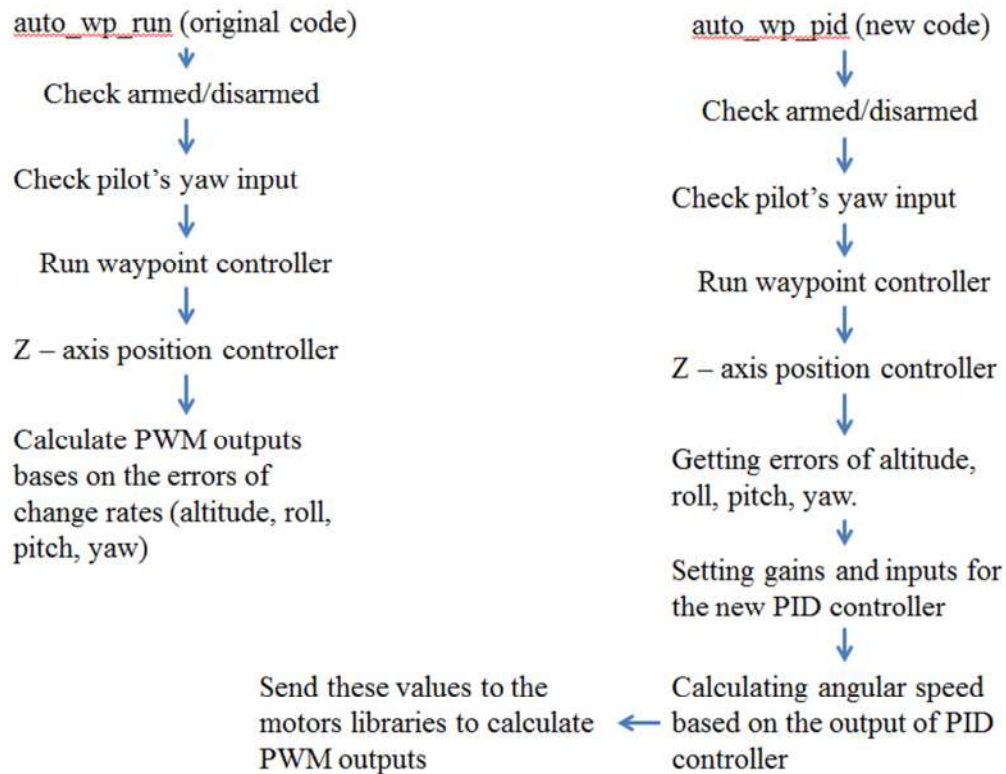


Figure 3-12: Old command (left) and new one (right) with the tasks they will carry on. At this stage, the integration is considered as done. Along with the modifications in the simulation code, SITL simulation is ready to test the new PID controller. In order to get back to the original PID controller, developers only need to undo the setting in step 3 and 4. Changing between two controllers gives developers the ability to compare the results to see the benefits and drawbacks of each controller.

3.5 New PID controller results

Figure 3-13 and 3-14 introduces the flight plans used in this report. Each of these missions will test not only the controllability of the new controller but also its ability to trace the flight path of the navigation system of ArduPilot.



Figure 3-13: First flight plan



Figure 3-14: Second flight plan

The PID gains for this test can be seen in **table 3** below, the values with the uppercase name is the name of the control constant related to the original PID controller. By using these variables, developers can change the PID gains of the new controller by using the ground control station and MAVLink protocol. More detail about this changing can be found in report #2 [6].

Table 3: PID gains

Altitude PID controller		
Gains	Setting value	Default value
P	THROTTLE_ACCEL_P	0.5
I	THROTTLE_ACCEL_I/20.0	0.05

D	THROTTLE_ACCEL_D+2.0	2.0
Phi (Roll) PID controller		
Gains	Setting value	Default value
P	RATE_ROLL_P/5.0	0.03
I	RATE_ROLL_I	0.1
D	RATE_ROLL_D + 2.0	2.004
Theta (Pitch) PID controller		
Gains	Setting value	Default value
P	RATE_PITCH_P/5.0	0.03
I	RATE_PITCH_I	0.1
D	RATE_PITCH_D + 1.0	1.004
Psi (Yaw) PID controller		
Gains	Setting value	Default value
P	RATE_YAW_P/10.0	0.02
I	RATE_YAW_I	0.02
D	RATE_YAW_D + 1.0	1.0

Figure 3-15 and **figure 3-16** shows a SITL simulation test in action, the blue line in the figure is the flight path the quadricopter has just moved through, comparing with the yellow is the desired flight path in the mission. As can be seen, although in **figure 3-17** and **3-18**, there are some differences between the blue and the yellow line, the simulated flight path and the desired one is very close with each other.

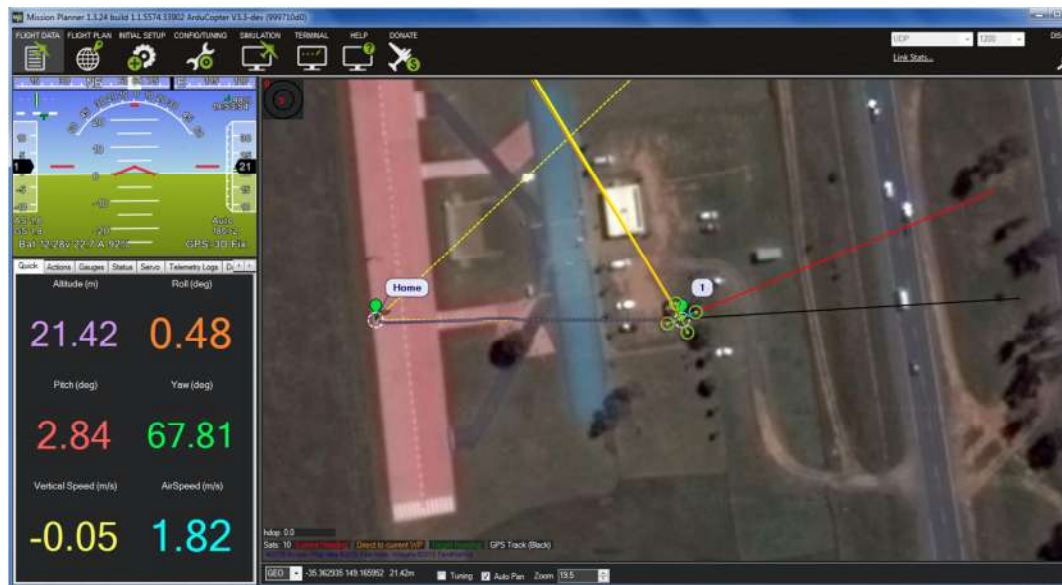


Figure 3-15: A SITL simulation with new PID controller

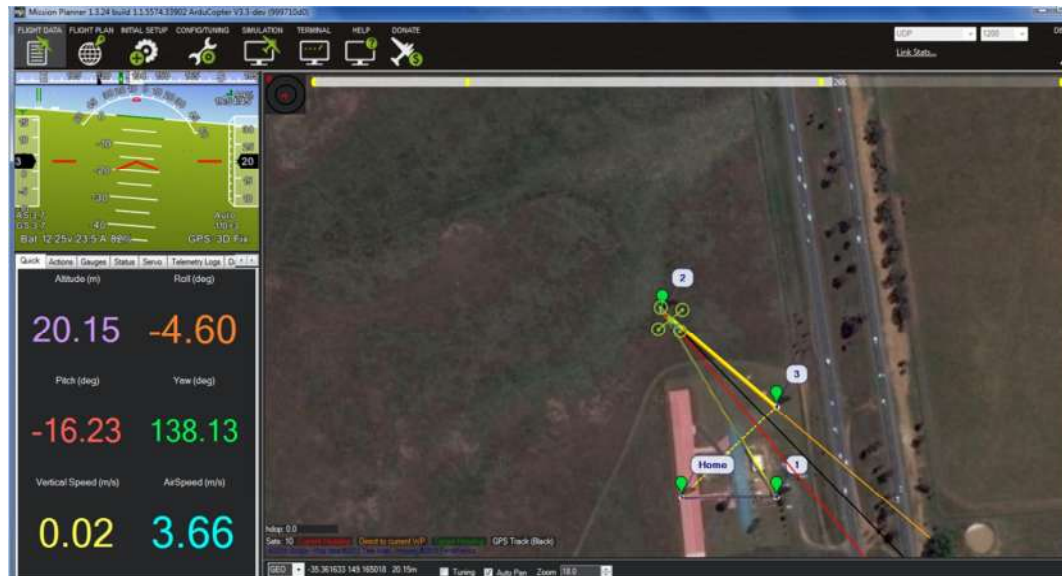


Figure 3-16: A SITL simulation with new PID controller (2)

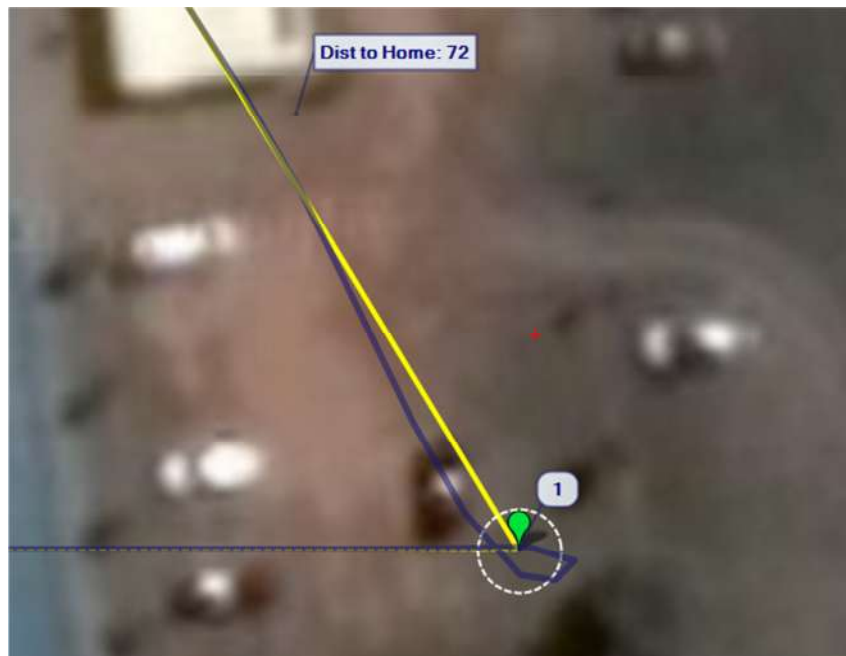


Figure 3-17: Overshoot and re-stabilize with new PID controller

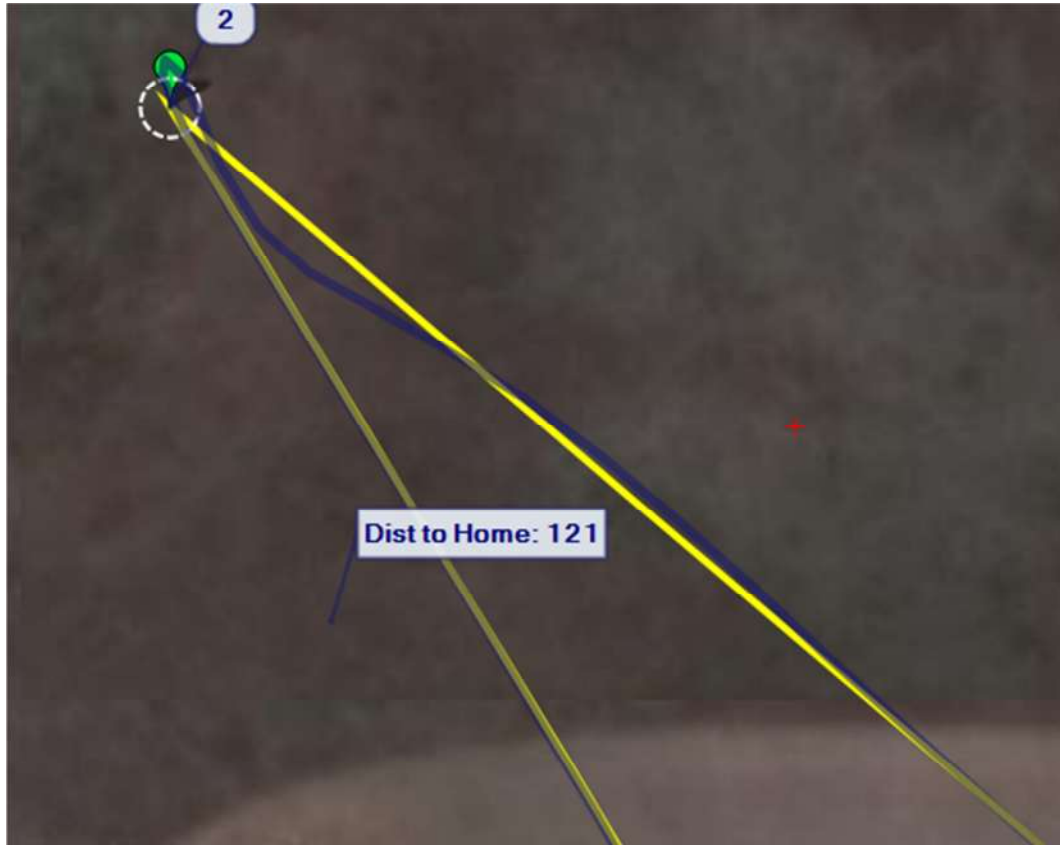


Figure 3-18: Overshoot and re-stabilize with new PID controller (2)

Using the analyzing tool provided by Mission Planner to create the data file for Matlab, **figure 3-19** introduces the differences between desired pitch (generate by the differences between current position and position of position target) and the response pitch generated by the response of the quadricopter in simulation. In this figure, the horizontal axis is the time in second; meanwhile, the vertical one represents degree value. As can be seen from that figure, although there are differences between the desired pitch (red line) and the result pitch (blue line), both of them share a familiar tendency, going up and down together. The two main different period is the one at which the quadricopter has reached its current destination and the navigation system is try to moving on to the next one. At this time, the quadricopter not only need to change its heading as fast as possible but also change its current moving direction. To understand more about this, the differences in roll and yaw also are also introduced in **figure 3-20** and **fig 3-21** respectively.

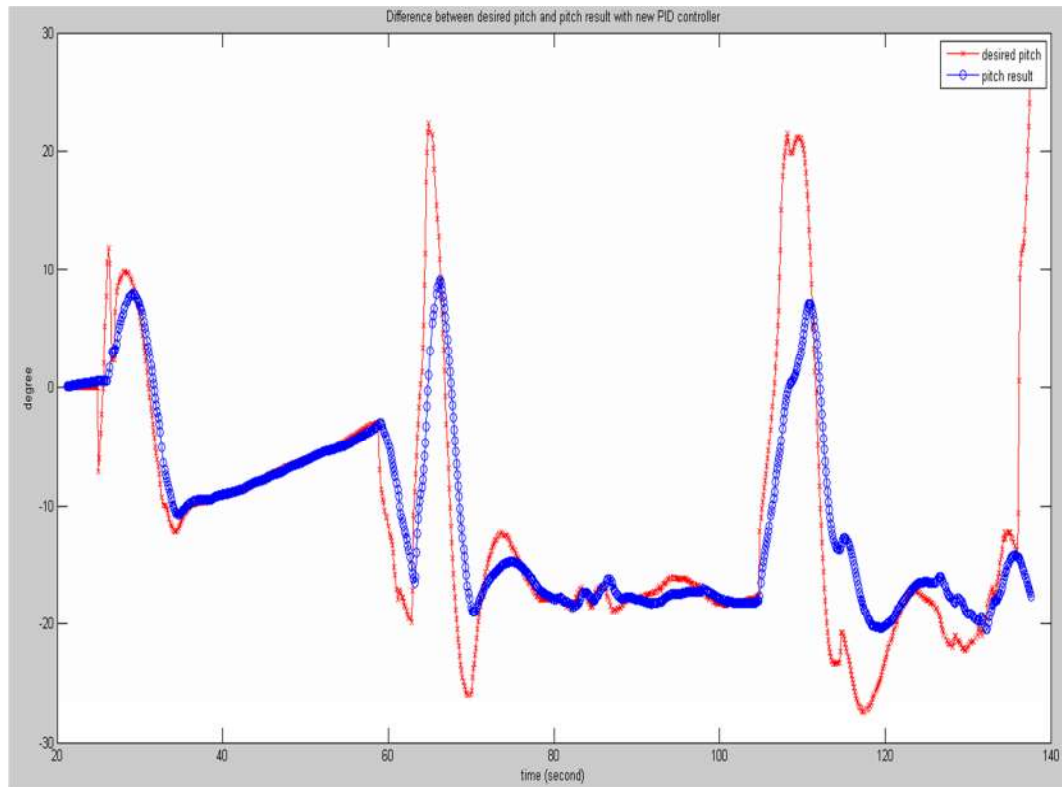


Figure 3-19: Difference between desired pitch and pitch result with new PID controller

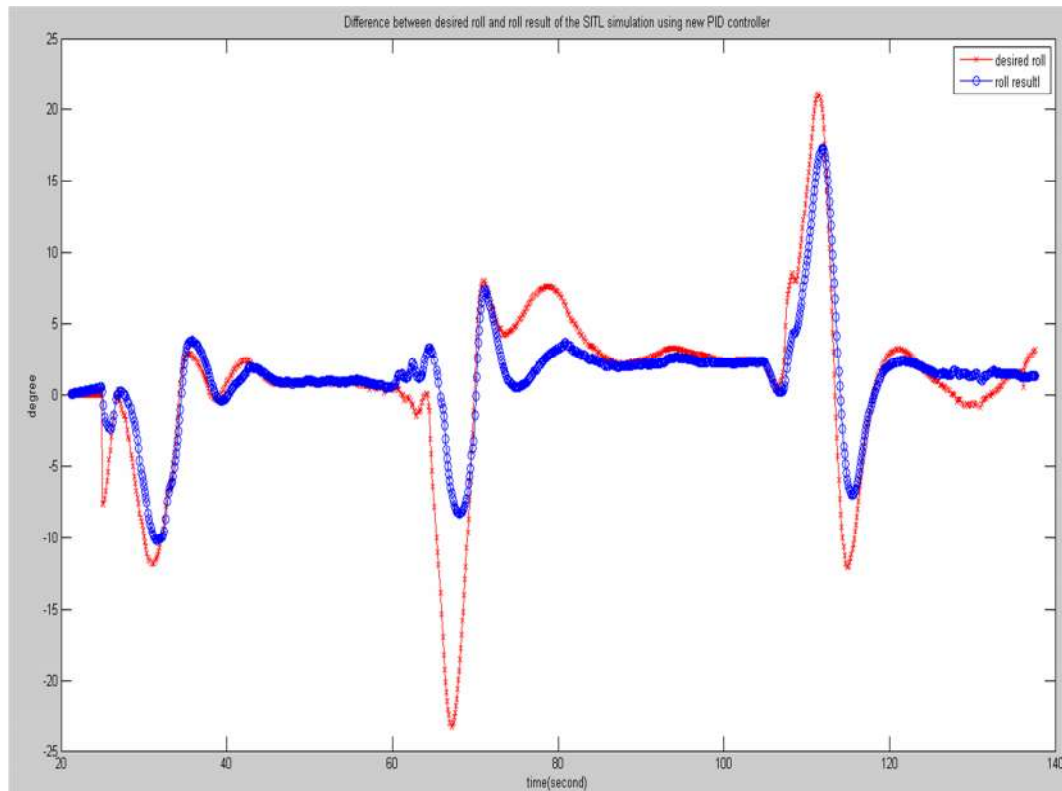


Figure 3-20: Difference between desired roll and roll result with new PID controller

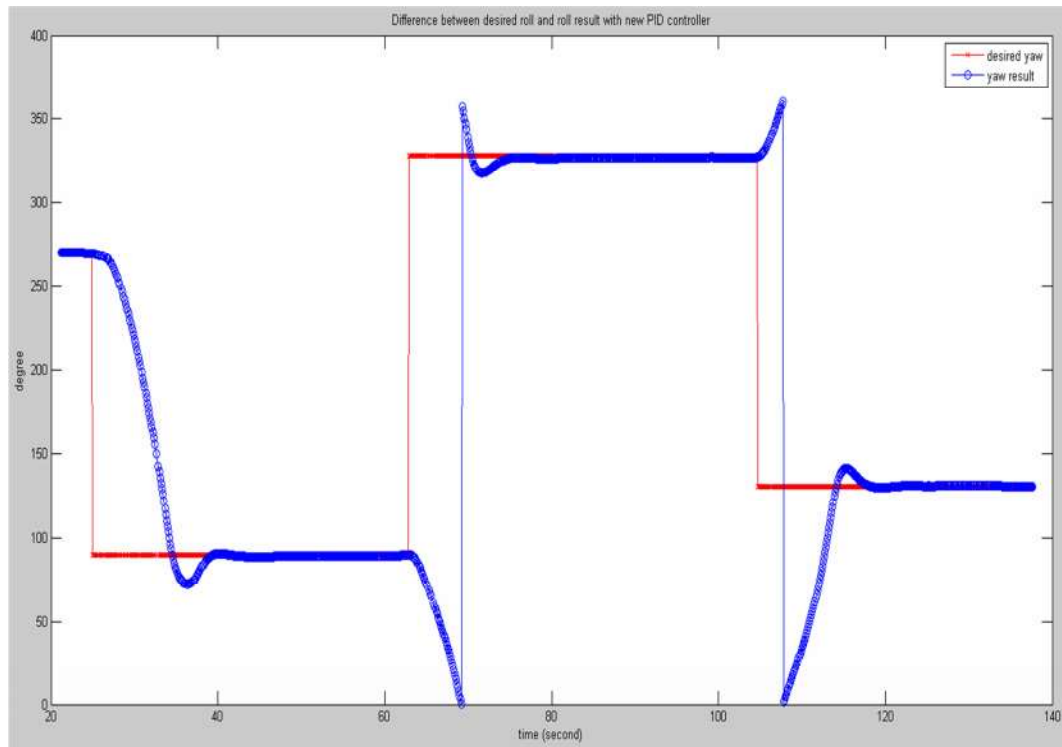


Figure 3-21: Difference between desired yaw and yaw result with new PID controller

In order to see how good the new PID controller is, the above results will be compared with the results from original PID controller.

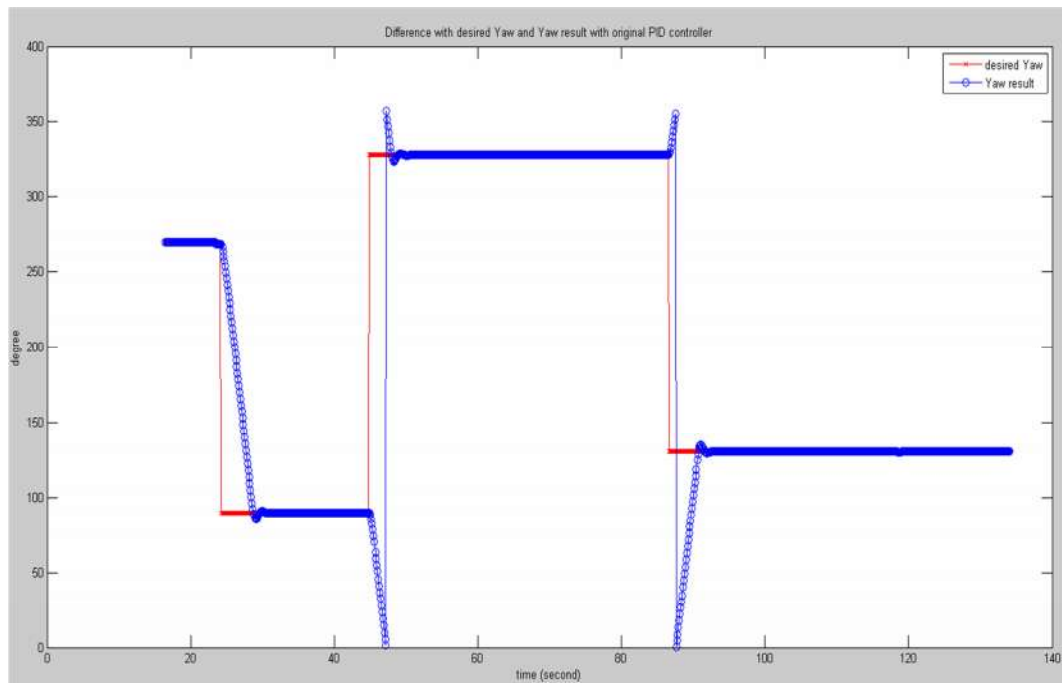


Figure 3-22: Difference between desired Yaw and Yaw result with original PID controller

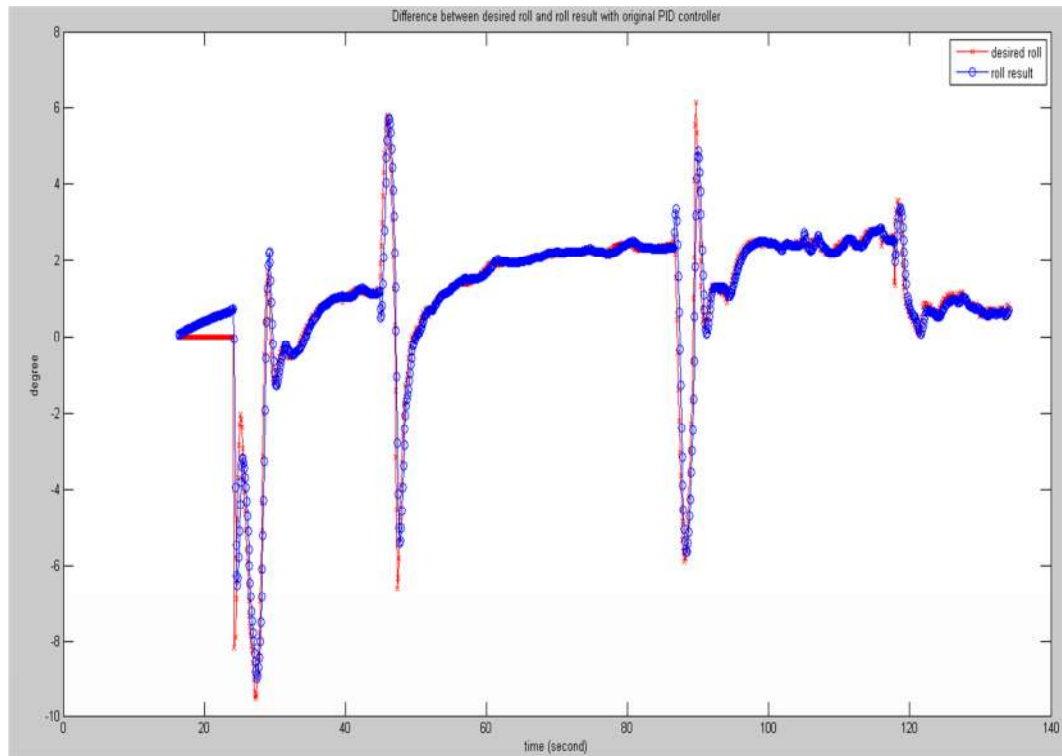


Figure 3-23: Difference between desired Roll and Roll result with original PID controller

Figure 3-22 and **figure 3-23** shows the differences between desired and real Yaw as well as Roll with the default PID controller of ArduPilot. From the above figures of both new PID controller and old one, it can be concluded that:

- The new PID controller and its generated code can be embedded into the framework of ArduPilot. The procedure mentioned above has successfully integrated new controller into ArduPilot without changing its architecture.
- The PID gains mentioned above still need to be optimized to reduce the overshoot as well as the settling time. However, because of the virtual quadricopter in this simulation is not identical with the real one, the adjustments here is not very important since the modification process will have to be done all over again with the real hardware. The utilization of SITL simulation here just make sure that the new controller work properly and ready to test with the real one.

To be sum up, the results created by Software in the Loop simulation proves that the integration of new controller into ArduPilot can be done without changing so much the original code. With the above results, the remaining work, although there are various measurements and calculation must be done (for the motor output signal, PID gains, etc), is to prepare the real hardware for a real flight with the firmware using new PID controller.

References

- [1] A.Q.Nguyễn, "ArduPilot, Overview, basic structure, problems and solutions," Poitiers, Intership 2015.
- [2] Flight Modes. [Online]. <http://copter.ardupilot.com/wiki/flying-arducopter/flight-modes/>
- [3] A.Castillo Benito, "Flight Control and Navigation of a Quadcopter," ISAE-ENSMA, Poitiers, Internship Report 2014.
- [4] Martin Gomez. (2001, Nov) Hardware-in-the-Loop Simulation. [Online]. <http://www.embedded.com/design/prototyping-and-development/4024865/Hardware-in-the-Loop-Simulation>
- [5] SITL Simulator (Software in the Loop). [Online]. <http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/>
- [6] A.Q.Nguyễn, "MAVLink, communication protocol for unmanned vehicle," Internship 2015.
- [7] Setting up SITL on Windows. [Online]. <http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/setting-up-sitl-on-windows/>
- [8] Setting up SITL on Linux. [Online]. <http://dev.ardupilot.com/wiki/simulation-2/sitl-simulator-software-in-the-loop/setting-up-sitl-on-linux/>

[loop/setting-up-sitl-on-linux/](#)

- [9] N.Đạt BUI, "Embedded System for Quadricopter," Ho Chi Minh City University of Technology, Internship Report 2014.
- [10] THE GENE-AUTO PROJECT. [Online]. <http://geneauto.gforge.enseeiht.fr/>