UNLV INTERNSHIP

Emergency response UAVs

Matthieu Cambusier

Tutor

Mr Venkatesan Muthukumar

May 2018 – July 2018

Thanking :

I would like to thank first the UNLV which allow me to do my internship within the univerisity.

I am greatful to Mr Muthukumar who accepted to be my tutor and his good advices that helped me during my intern.

I also want to thank Mr Cho for his help while we were filling out the administrative papers for the UNLV.

Abstract :

The purpose of this project is to detect someone screaming for help with a drone to rescue him.

To do this, we calibrated the drone with the pixhawk 4 so that it could fly properly. Then, we collected some data (voice, bullet, car, wind …) and created an algorithm to mix them and apply denoising on this.

Table of contents

Introduction :

As part of my engineering training, a minimum stay of 3 months abroad is required to obtain the diploma. This is why I choosed to do an internship of 3 months at the University of Nevada Las Vegas (UNLV).

My internship tooked place in the field of the aeronautic by working on a drone.

A drone is an unmanned "robotic" vehicle that can be remotely or autonomously controlled. Drones are used for many consumer, industrial and military use cases and applications :

- Aerial photography/video
- Carrying cargo
- Racing
- Search
- Surveying

Nowadays, the security is very important and when someone is in danger, it is important to locate this person very quickly to rescue him/her so that he/she can be treat as quickly as possible.

To do that, the use of a drone is a good solution. The aim of this project is that a drone detect someone calling « Get help » despite the other sound around.

In a first part I will introduce the materials used for this project and how they work. The second part will be about all the technical and code made.

Material and tasks :

I. Equipment :

1. Quadcopter :

The quadcopter was made by *Skyworks Aerial Systems*.

The quadcopter is a cheap kit and you have to assemble it. Fortunately for me, the dorne was already asssembled.

Picture :Drone assembled

To flight the drone you need to connect it with a Pixhawk.

2. Pixhawk :

A) Presentation :

Pixhawk is an independent open-hardware project that aims to provide the standard for readily-available, hiqh-quality and low-cost autopilot hardware designs for the academic, hobby and developer communities. It features advanced processor and sensor technology from ST Microelectronics and a NuttX real-time operating system, delivering performance, flexibility, and reliability for controlling any autonomous vehicle.

Manufacturers have created many different boards, but in this project we worked with the « Pixhawk 4 ».

B) Pixhawk 4 :

Pixhawk 4 is an advanced autopilot designed and made in collaboration with Holybro and the PX4 team. PX4 all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles.

It is optimized to run PX4 version 1.7, suitable for academic and commercial developers.

It is based on the Pixhawk-project FMUv5 open hardware design and runs PX4 on the NuttX OS.

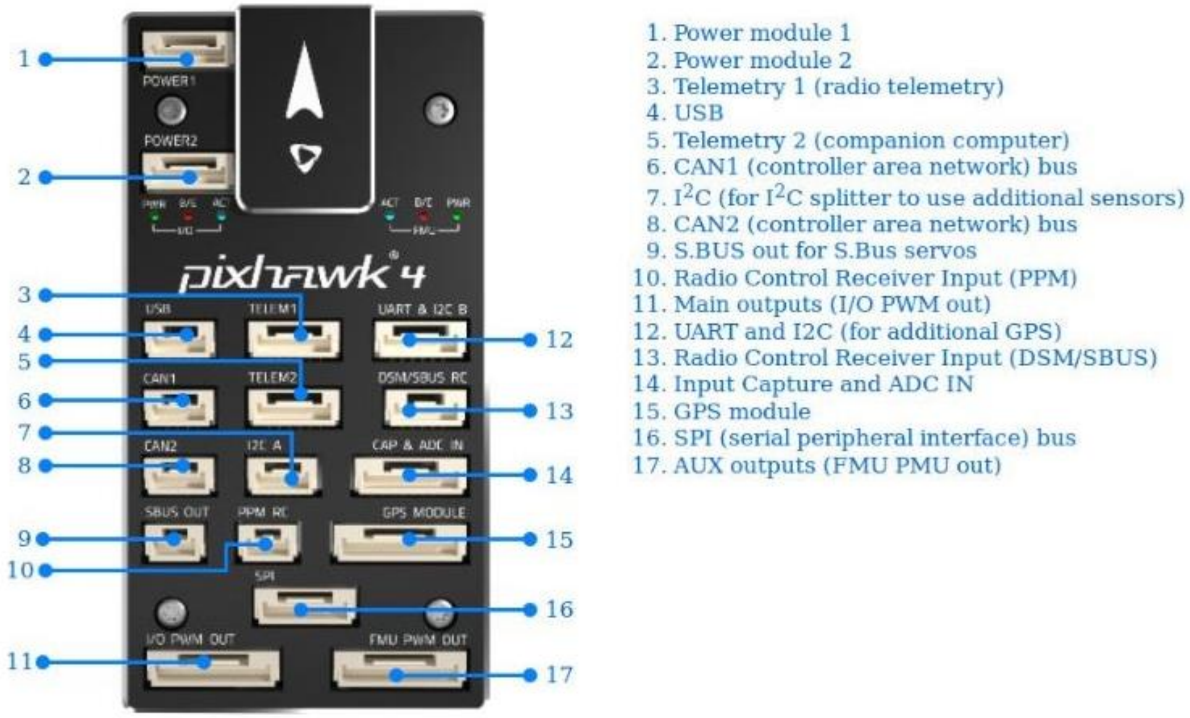


Figure : Pixhawk connectors

Figure : Pixhawk connectors

PX4 uses sensors to determine vehicle state (position/altitude, heading, speed, airspeed, orientation, rates of rotation in different directions, battery level, etc.). For the drone’s flight, I calibrated these sensors to flight the drone properly.

C) Hardware installation :

The following picture shows how to connect the different components to the pixhawk :

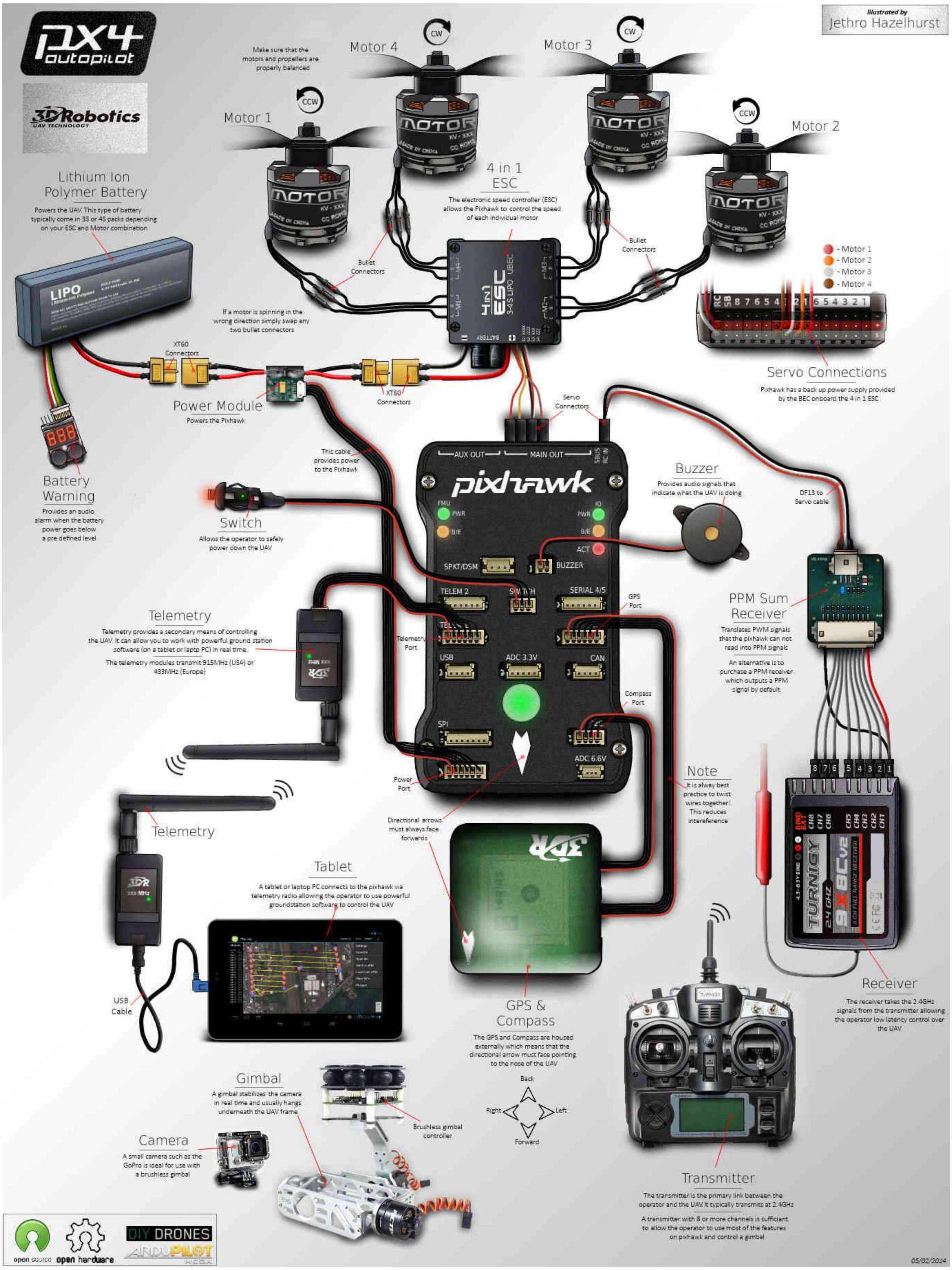


Figure :Pixhawk connected to the other equipments

3. Digital voice recorder USB :

To record the data, we used an USB key that record the voice.

Figure :USB key voice recorder

Don’t forget to check if the battery is charged to use the microphone.

II. Tasks :

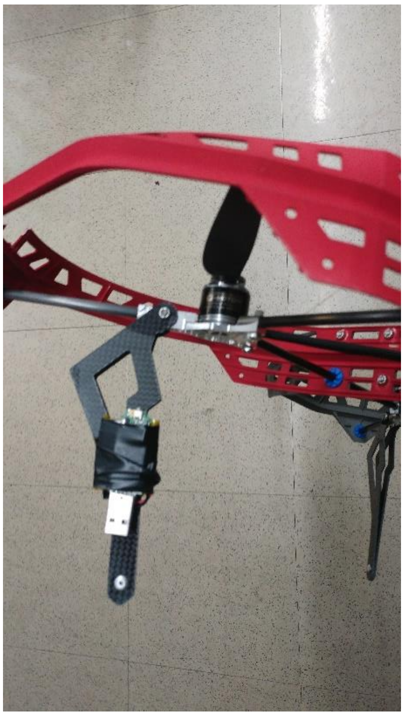
1. Quadcopter flight :

Once the drone calibration done, I did a flight with the drone. All the explications for the calibration of the drone and the drone's flight are in the slides « tutorial_QGroundController ».

Picture : Flight with the quadcopter

2. Collecting noise :

To collect the data, I scotched the USB key on the drone but I had to scotched near to the helix.




Picture : USB key scotched to the drone

For the recording, slide the button on « ON ». You will see a red light and after it will turn into a blue light. At this moment, this means that USB is recording.

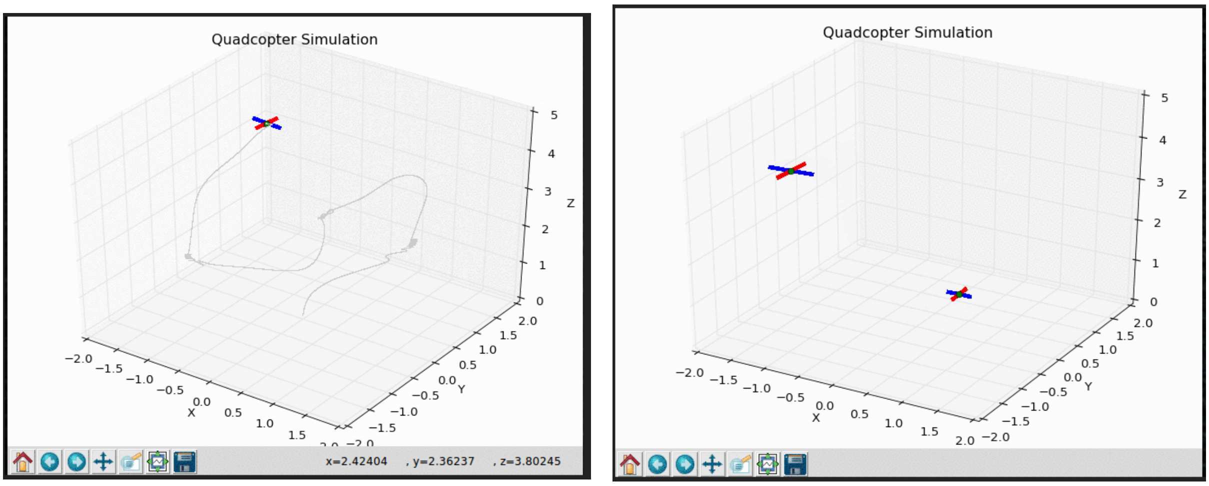
3. Code :

A) Simulation :

For the simulation, I have 4 classes :

- The controller, which control the drone.
- The GUI, it shows the drone flying.
- The quadcopter where the different equation of state are defined.
- The simulation, we can simulate one or more drone(s).

The code will be in the annex part.



Picture : Quadcopter(s) simulation

B) Denoising algorithm :

a: Python :

Once the data recorded, I had to apply a denoising algortihm.

First, I did a Python code

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 13 19:53:19 2018

@author: matth
"""

from scipy.io import wavfile
from scipy.signal import resample

#import StringIO
import base64
import struct

# change to the shogun-data directoy
import os
os.chdir('C:/Users/matth/Desktop/sound')

#%pylab inline
import pylab as pl
import numpy as np

from IPython.core.display import HTML
from shogun.Features  import RealFeatures
from shogun.Converter import Jade
```

Picture : Python code

```python
def load_wav(filename,samplerate=44100):

    # loading the file
    rate, data = wavfile.read(filename)

    # conversion of the stereo audio into a mono audio
    if len(data.shape) > 1:
        data = data[:,0]/2 + data[:,1]/2

    # re-interpose samplerate
    ratio = float(samplerate) / float(rate)
    data = resample(data, len(data) * ratio)

    return samplerate, data.astype(np.int16)
```

Picture : Python code

```
def wavPlayer(data, rate):

    buffer = six.moves.StringIO()
    buffer.write(b'RIFF')
    buffer.write(b'\x00\x00\x00\x00')
    buffer.write(b'WAVE')

    buffer.write(b'fmt ')
    if data.ndim == 1:
        noc = 1
    else:
        noc = data.shape[1]
    bits = data.dtype.itemsize * 8
    sbytes = rate*(bits // 8)*noc
    ba = noc * (bits // 8)
    buffer.write(struct.pack('<ihHIIHH', 16, 1, noc, rate, sbytes, ba, bits))

    # data block
    buffer.write(b'data')
    buffer.write(struct.pack('<i', data.nbytes))

    if data.dtype.byteorder == '>' or (data.dtype.byteorder == '=' and sys.byteorder == 'big'):
        data = data.byteswap()

    buffer.write(data.tostring())
    # return buffer.getvalue()
    # Determine file size and place it in correct
    # position at start of the file.
    size = buffer.tell()
    buffer.seek(4)
    buffer.write(struct.pack('<i', size-8))

    val = buffer.getvalue()

    src = """
```

Picture : Python code

```
    src = """
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Simple Test</title>
    </head>

    <body>
    <audio controls="controls" style="width:600px" >
      <source controls src="data:audio/wav;base64,{base64}" type="audio/wav" />
      Your browser does not support the audio element.
    </audio>
    </body>
    """.format(base64=base64.encodestring(val))
    display(HTML(src))
```

Picture : Python code

```
# loading the first file
fs1,s1 = load_wav('ak471.wav')
# plot the audio file
pl.figure(figsize=(7,2))
pl.plot(s1)
pl.title('Signal 1')
pl.show()
# play the audio file
wavPlayer(s1, fs1)

# loading the second file
fs2,s2 = load_wav('siren.wav')
# plot the audio file
pl.figure(figsize=(6.75,2))
pl.plot(s2)
pl.title('Signal 2')
pl.show()
# play the audio file
wavPlayer(s2, fs2)


# Adjust for different clip lengths
```

Picture : Python code

```
# Adjust for different clip lengths
fs = fs1
length = max([len(s1), len(s2)])

s1.resize((length,1), refcheck=False)
s2.resize((length,1), refcheck=False)

S = (np.c_[s1, s2]).T      #transposition for having an input source matrix with the right size in accord to the chosen mixing matrix'

# Mixing Matrix
A = np.array([[1, 0.5],
              [0.5, 1]])
print ('Mixing Matrix:')
print (A.round(2))

# Mixed Signals
```

Picture : Python code

```
# Mixed Signals
X = np.dot(A,S)

# Exploring Mixed Signals
for i in range(X.shape[0]):
    pl.figure(figsize=(6.75,2))
    pl.plot((X[i]).astype(np.int16))
    pl.title('Mixed Signal %d' % (i+1))
    pl.show()
    wavPlayer((X[i]).astype(np.int16), fs)

# Convert to features for shogun
mixed_signals = RealFeatures((X).astype(np.float64))

# Separating with JADE
jade = Jade()
signals = jade.apply(mixed_signals)

S_ = signals.get_feature_matrix()

A_ = jade.get_mixing_matrix()
A_ = A_ / A_.sum(axis=0)

print ('Estimated Mixing Matrix:')
print (A_)

# Show separation results

# Separated Signal i
gain = 4000
for i in range(S_.shape[0]):
    pl.figure(figsize=(6.75,2))
    pl.plot((gain*S_[i]).astype(np.int16))
    pl.title('Separated Signal %d' % (i+1))
    pl.show()
    wavPlayer((gain*S_[i]).astype(np.int16), fs)
```

Picture : Python code

Unfortunately, the code doesn't work because of the shogun module

III. Conclusion :

During this internship, I learnt a lot about the drone (the operation, the theory …).

I also learned more about the embedded systems, the autonomous vehicle and how works the denoising.

This internship allow me to improve my skills in communication and to experience an other culture

IV. Annex :

1. Python code for the simulation :

A) Controller :

```python
import numpy as np
import math
import time
import threading

class Controller_PID_Point2Point():
    def __init__(self, get_state, get_time, actuate_motors, params, quad_identifier):
        self.quad_identifier = quad_identifier
        self.actuate_motors = actuate_motors
        self.get_state = get_state
        self.get_time = get_time
        self.MOTOR_LIMITS = params['Motor_limits']
        self.TILT_LIMITS = [(params['Tilt_limits'][0]/180.0)*3.14,(params['Tilt_limits'][1]/180.0)*3.14]
        self.YAW_CONTROL_LIMITS = params['Yaw_Control_Limits']
        self.Z_LIMITS = [self.MOTOR_LIMITS[0]+params['Z_XY_offset'],self.MOTOR_LIMITS[1]-params['Z_XY_offset']]
        self.LINEAR_P = params['Linear_PID']['P']
        self.LINEAR_I = params['Linear_PID']['I']
        self.LINEAR_D = params['Linear_PID']['D']
        self.LINEAR_TO_ANGULAR_SCALER = params['Linear_To_Angular_Scaler']
        self.YAW_RATE_SCALER = params['Yaw_Rate_Scaler']
        self.ANGULAR_P = params['Angular_PID']['P']
        self.ANGULAR_I = params['Angular_PID']['I']
        self.ANGULAR_D = params['Angular_PID']['D']
        self.xi_term = 0
        self.yi_term = 0
        self.zi_term = 0
        self.thetai_term = 0
        self.phii_term = 0
        self.gammai_term = 0
        self.thread_object = None
        self.target = [0,0,0]
        self.yaw_target = 0.0
        self.run = True

    def wrap_angle(self,val):
        return( ( val + np.pi) % (2 * np.pi ) - np.pi )

def update(self):
    [dest_x,dest_y,dest_z] = self.target
    [x,y,z,x_dot,y_dot,z_dot,theta,phi,gamma,theta_dot,phi_dot,gamma_dot] = self.get_state(self.quad_identifier)
    x_error = dest_x-x
    y_error = dest_y-y
    z_error = dest_z-z
    self.xi_term += self.LINEAR_I[0]*x_error
    self.yi_term += self.LINEAR_I[1]*y_error
    self.zi_term += self.LINEAR_I[2]*z_error
    dest_x_dot = self.LINEAR_P[0]*(x_error) + self.LINEAR_D[0]*(-x_dot) + self.xi_term
    dest_y_dot = self.LINEAR_P[1]*(y_error) + self.LINEAR_D[1]*(-y_dot) + self.yi_term
    dest_z_dot = self.LINEAR_P[2]*(z_error) + self.LINEAR_D[2]*(-z_dot) + self.zi_term
    throttle = np.clip(dest_z_dot,self.Z_LIMITS[0],self.Z_LIMITS[1])
    dest_theta = self.LINEAR_TO_ANGULAR_SCALER[0]*(dest_x_dot*math.sin(gamma)-dest_y_dot*math.cos(gamma))
    dest_phi = self.LINEAR_TO_ANGULAR_SCALER[1]*(dest_x_dot*math.cos(gamma)+dest_y_dot*math.sin(gamma))
    dest_gamma = self.yaw_target
    dest_theta,dest_phi = np.clip(dest_theta,self.TILT_LIMITS[0],self.TILT_LIMITS[1]),np.clip(dest_phi,self.TILT_LIMITS[0],self.TILT_LIMITS[1])
    theta_error = dest_theta-theta
    phi_error = dest_phi-phi
    gamma_dot_error = (self.YAW_RATE_SCALER*self.wrap_angle(dest_gamma-gamma)) - gamma_dot
    self.thetai_term += self.ANGULAR_I[0]*theta_error
    self.phii_term += self.ANGULAR_I[1]*phi_error
    self.gammai_term += self.ANGULAR_I[2]*gamma_dot_error
    x_val = self.ANGULAR_P[0]*(theta_error) + self.ANGULAR_D[0]*(-theta_dot) + self.thetai_term
    y_val = self.ANGULAR_P[1]*(phi_error) + self.ANGULAR_D[1]*(-phi_dot) + self.phii_term
    z_val = self.ANGULAR_P[2]*(gamma_dot_error) + self.gammai_term
    z_val = np.clip(z_val,self.YAW_CONTROL_LIMITS[0],self.YAW_CONTROL_LIMITS[1])
    m1 = throttle + x_val + z_val
    m2 = throttle + y_val - z_val
    m3 = throttle - x_val + z_val
    m4 = throttle - y_val - z_val
    M = np.clip([m1,m2,m3,m4],self.MOTOR_LIMITS[0],self.MOTOR_LIMITS[1])
    self.actuate_motors(self.quad_identifier,M)

def update_target(self,target):
    self.target = target

def update_yaw_target(self,target):
    self.yaw_target = self.wrap_angle(target)
```

```python
    def thread_run(self,update_rate,time_scaling):
        update_rate = update_rate*time_scaling
        last_update = self.get_time()
        while(self.run==True):
            time.sleep(0)
            self.time = self.get_time()
            if (self.time - last_update).total_seconds() > update_rate:
                self.update()
                last_update = self.time

    def start_thread(self,update_rate=0.005,time_scaling=1):
        self.thread_object = threading.Thread(target=self.thread_run,args=(update_rate,time_scaling))
        self.thread_object.start()

    def stop_thread(self):
        self.run = False
```

```python
class Controller_PID_Velocity(Controller_PID_Point2Point):
    def update(self):
        [dest_x,dest_y,dest_z] = self.target
        [x,y,z,x_dot,y_dot,z_dot,theta,phi,gamma,theta_dot,phi_dot,gamma_dot] = self.get_state(self.quad_identifier)
        x_error = dest_x-x_dot
        y_error = dest_y-y_dot
        z_error = dest_z-z
        self.xi_term += self.LINEAR_I[0]*x_error
        self.yi_term += self.LINEAR_I[1]*y_error
        self.zi_term += self.LINEAR_I[2]*z_error
        dest_x_dot = self.LINEAR_P[0]*(x_error) + self.LINEAR_D[0]*(-x_dot) + self.xi_term
        dest_y_dot = self.LINEAR_P[1]*(y_error) + self.LINEAR_D[1]*(-y_dot) + self.yi_term
        dest_z_dot = self.LINEAR_P[2]*(z_error) + self.LINEAR_D[2]*(-z_dot) + self.zi_term
        throttle = np.clip(dest_z_dot,self.Z_LIMITS[0],self.Z_LIMITS[1])
        dest_theta = self.LINEAR_TO_ANGULAR_SCALER[0]*(dest_x_dot*math.sin(gamma)-dest_y_dot*math.cos(gamma))
        dest_phi = self.LINEAR_TO_ANGULAR_SCALER[1]*(dest_x_dot*math.cos(gamma)+dest_y_dot*math.sin(gamma))
        dest_gamma = self.yaw_target
        dest_theta,dest_phi = np.clip(dest_theta,self.TILT_LIMITS[0],self.TILT_LIMITS[1]),np.clip(dest_phi,self.TILT_LIMITS[0],self.TILT_LIMITS[1])
        theta_error = dest_theta-theta
        phi_error = dest_phi-phi
        gamma_dot_error = (self.YAW_RATE_SCALER*self.wrap_angle(dest_gamma-gamma)) - gamma_dot
        self.thetai_term += self.ANGULAR_I[0]*theta_error
        self.phii_term += self.ANGULAR_I[1]*phi_error
        self.gammai_term += self.ANGULAR_I[2]*gamma_dot_error
        x_val = self.ANGULAR_P[0]*(theta_error) + self.ANGULAR_D[0]*(-theta_dot) + self.thetai_term
        y_val = self.ANGULAR_P[1]*(phi_error) + self.ANGULAR_D[1]*(-phi_dot) + self.phii_term
        z_val = self.ANGULAR_P[2]*(gamma_dot_error) + self.gammai_term
        z_val = np.clip(z_val,self.YAW_CONTROL_LIMITS[0],self.YAW_CONTROL_LIMITS[1])
        m1 = throttle + x_val + z_val
        m2 = throttle + y_val - z_val
        m3 = throttle - x_val + z_val
        m4 = throttle - y_val - z_val
        M = np.clip([m1,m2,m3,m4],self.MOTOR_LIMITS[0],self.MOTOR_LIMITS[1])
        self.actuate_motors(self.quad_identifier,M)
```

B) GUI :

```
import numpy as np
import math
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as Axes3D
import sys

class GUI():
    # 'quad_list' is a dictionary of format: quad_list = {'quad_1_name':{'posit
    def __init__(self, quads):
        self.quads = quads
        self.fig = plt.figure()
        self.ax = Axes3D.Axes3D(self.fig)
        self.ax.set_xlim3d([-2.0, 2.0])
        self.ax.set_xlabel('X')
        self.ax.set_ylim3d([-2.0, 2.0])
        self.ax.set_ylabel('Y')
        self.ax.set_zlim3d([0, 5.0])
        self.ax.set_zlabel('Z')
        self.ax.set_title('Quadcopter Simulation')
        self.init_plot()
        self.fig.canvas.mpl_connect('key_press_event', self.keypress_routine)

    def rotation_matrix(self,angles):
        ct = math.cos(angles[0])
        cp = math.cos(angles[1])
        cg = math.cos(angles[2])
        st = math.sin(angles[0])
        sp = math.sin(angles[1])
        sg = math.sin(angles[2])
        R_x = np.array([[1,0,0],[0,ct,-st],[0,st,ct]])
        R_y = np.array([[cp,0,sp],[0,1,0],[-sp,0,cp]])
        R_z = np.array([[cg,-sg,0],[sg,cg,0],[0,0,1]])
        R = np.dot(R_z, np.dot( R_y, R_x ))
        return R
```

```
def init_plot(self):
    for key in self.quads:
        self.quads[key]['l1'], = self.ax.plot([],[],[],color='blue',linewidth=3,antialiased=False)
        self.quads[key]['l2'], = self.ax.plot([],[],[],color='red',linewidth=3,antialiased=False)
        self.quads[key]['hub'], = self.ax.plot([],[],[],marker='o',color='green', markersize=6,antialiased=False)

def update(self):
    for key in self.quads:
        R = self.rotation_matrix(self.quads[key]['orientation'])
        L = self.quads[key]['L']
        points = np.array([ [-L,0,0], [L,0,0], [0,-L,0], [0,L,0], [0,0,0], [0,0,0] ]).T
        points = np.dot(R,points)
        points[0,:] += self.quads[key]['position'][0]
        points[1,:] += self.quads[key]['position'][1]
        points[2,:] += self.quads[key]['position'][2]
        self.quads[key]['l1'].set_data(points[0,0:2],points[1,0:2])
        self.quads[key]['l1'].set_3d_properties(points[2,0:2])
        self.quads[key]['l2'].set_data(points[0,2:4],points[1,2:4])
        self.quads[key]['l2'].set_3d_properties(points[2,2:4])
        self.quads[key]['hub'].set_data(points[0,5],points[1,5])
        self.quads[key]['hub'].set_3d_properties(points[2,5])
    plt.pause(0.000000000000001)
```

```
def keypress_routine(self,event):
    sys.stdout.flush()
    if event.key == 'x':
        y = list(self.ax.get_ylim3d())
        y[0] += 0.2
        y[1] += 0.2
        self.ax.set_ylim3d(y)
    elif event.key == 'w':
        y = list(self.ax.get_ylim3d())
        y[0] -= 0.2
        y[1] -= 0.2
        self.ax.set_ylim3d(y)
    elif event.key == 'd':
        x = list(self.ax.get_xlim3d())
        x[0] += 0.2
        x[1] += 0.2
        self.ax.set_xlim3d(x)
    elif event.key == 'a':
        x = list(self.ax.get_xlim3d())
        x[0] -= 0.2
        x[1] -= 0.2
        self.ax.set_xlim3d(x)
```

C) Quadcopter :

```python
import numpy as np
import math
import scipy.integrate
import time
import datetime
import threading

class Propeller():
    def __init__(self, prop_dia, prop_pitch, thrust_unit='N'):
        self.dia = prop_dia
        self.pitch = prop_pitch
        self.thrust_unit = thrust_unit
        self.speed = 0 #RPM
        self.thrust = 0

    def set_speed(self,speed):
        self.speed = speed
        # From http://www.electricrcaircraftguy.com/2013/09/propeller-static-dynamic-thrust-equation.html
        self.thrust = 4.392e-8 * self.speed * math.pow(self.dia,3.5)/(math.sqrt(self.pitch))
        self.thrust = self.thrust*(4.23e-4 * self.speed * self.pitch)
        if self.thrust_unit == 'Kg':
            self.thrust = self.thrust*0.101972
```

```python
class Quadcopter():
    # State space representation: [x y z x_dot y_dot z_dot theta phi gamma theta_dot phi_dot gamma_dot]
    # From Quadcopter Dynamics, Simulation, and Control by Andrew Gibiansky
    def __init__(self,quads,gravity=9.81,b=0.0245):
        self.quads = quads
        self.g = gravity
        self.b = b
        self.thread_object = None
        self.ode =  scipy.integrate.ode(self.state_dot).set_integrator('vode',nsteps=500,method='bdf')
        self.time = datetime.datetime.now()
        for key in self.quads:
            self.quads[key]['state'] = np.zeros(12)
            self.quads[key]['state'][0:3] = self.quads[key]['position']
            self.quads[key]['state'][6:9] = self.quads[key]['orientation']
            self.quads[key]['m1'] = Propeller(self.quads[key]['prop_size'][0],self.quads[key]['prop_size'][1])
            self.quads[key]['m2'] = Propeller(self.quads[key]['prop_size'][0],self.quads[key]['prop_size'][1])
            self.quads[key]['m3'] = Propeller(self.quads[key]['prop_size'][0],self.quads[key]['prop_size'][1])
            self.quads[key]['m4'] = Propeller(self.quads[key]['prop_size'][0],self.quads[key]['prop_size'][1])
            # From Quadrotor Dynamics and Control by Randal Beard
            ixx=((2*self.quads[key]['weight']*self.quads[key]['r']**2)/5)+(2*self.quads[key]['weight']*self.quads[key]['L']**2)
            iyy=ixx
            izz=((2*self.quads[key]['weight']*self.quads[key]['r']**2)/5)+(4*self.quads[key]['weight']*self.quads[key]['L']**2)
            self.quads[key]['I'] = np.array([[ixx,0,0],[0,iyy,0],[0,0,izz]])
            self.quads[key]['invI'] = np.linalg.inv(self.quads[key]['I'])
        self.run = True
```

```python
    def rotation_matrix(self,angles):
        ct = math.cos(angles[0])
        cp = math.cos(angles[1])
        cg = math.cos(angles[2])
        st = math.sin(angles[0])
        sp = math.sin(angles[1])
        sg = math.sin(angles[2])
        R_x = np.array([[1,0,0],[0,ct,-st],[0,st,ct]])
        R_y = np.array([[cp,0,sp],[0,1,0],[-sp,0,cp]])
        R_z = np.array([[cg,-sg,0],[sg,cg,0],[0,0,1]])
        R = np.dot(R_z, np.dot( R_y, R_x ))
        return R

    def wrap_angle(self,val):
        return( ( val + np.pi) % (2 * np.pi ) - np.pi )
```

```python
def state_dot(self, time, state, key):
    state_dot = np.zeros(12)
    # The velocities(t+1 x_dots equal the t x_dots)
    state_dot[0] = self.quads[key]['state'][3]
    state_dot[1] = self.quads[key]['state'][4]
    state_dot[2] = self.quads[key]['state'][5]
    # The acceleration
    x_dotdot = np.array([0,0,-self.quads[key]['weight']*self.g]) + np.dot(self.rotation_matrix(self.quads[key]['state'][6:9]),np.array([0,0,(self.quads[key]['m1
    state_dot[3] = x_dotdot[0]
    state_dot[4] = x_dotdot[1]
    state_dot[5] = x_dotdot[2]
    # The angular rates(t+1 theta_dots equal the t theta_dots)
    state_dot[6] = self.quads[key]['state'][9]
    state_dot[7] = self.quads[key]['state'][10]
    state_dot[8] = self.quads[key]['state'][11]
    # The angular accelerations
    omega = self.quads[key]['state'][9:12]
    tau = np.array([self.quads[key]['L']*(self.quads[key]['m1'].thrust-self.quads[key]['m3'].thrust), self.quads[key]['L']*(self.quads[key]['m2'].thrust-self.qu
    omega_dot = np.dot(self.quads[key]['invI'], (tau - np.cross(omega, np.dot(self.quads[key]['I'],omega))))
    state_dot[9] = omega_dot[0]
    state_dot[10] = omega_dot[1]
    state_dot[11] = omega_dot[2]
    return state_dot
```

```python
    def update(self, dt):
        for key in self.quads:
            self.ode.set_initial_value(self.quads[key]['state'],0).set_f_params(key)
            self.quads[key]['state'] = self.ode.integrate(self.ode.t + dt)
            self.quads[key]['state'][6:9] = self.wrap_angle(self.quads[key]['state'][6:9])
            self.quads[key]['state'][2] = max(0,self.quads[key]['state'][2])

    def set_motor_speeds(self,quad_name,speeds):
        self.quads[quad_name]['m1'].set_speed(speeds[0])
        self.quads[quad_name]['m2'].set_speed(speeds[1])
        self.quads[quad_name]['m3'].set_speed(speeds[2])
        self.quads[quad_name]['m4'].set_speed(speeds[3])

    def get_position(self,quad_name):
        return self.quads[quad_name]['state'][0:3]

    def get_linear_rate(self,quad_name):
        return self.quads[quad_name]['state'][3:6]

    def get_orientation(self,quad_name):
        return self.quads[quad_name]['state'][6:9]

    def get_angular_rate(self,quad_name):
        return self.quads[quad_name]['state'][9:12]

    def get_state(self,quad_name):
        return self.quads[quad_name]['state']

    def set_position(self,quad_name,position):
        self.quads[quad_name]['state'][0:3] = position

    def set_orientation(self,quad_name,orientation):
        self.quads[quad_name]['state'][6:9] = orientation

    def get_time(self):
        return self.time
```

```python
def thread_run(self,dt,time_scaling):
    rate = time_scaling*dt
    last_update = self.time
    while(self.run==True):
        time.sleep(0)
        self.time = datetime.datetime.now()
        if (self.time-last_update).total_seconds() > rate:
            self.update(dt)
            last_update = self.time

def start_thread(self,dt=0.002,time_scaling=1):
    self.thread_object = threading.Thread(target=self.thread_run,args=(dt,time_scaling))
    self.thread_object.start()

def stop_thread(self):
    self.run = False
```

D) Simulation :

```python
import quadcopter,gui,controller
import signal
import sys
import argparse

# Constants
TIME_SCALING = 1.0 # Any positive number(Smaller is faster). 1.0->Real Time, 0.0->Run as fast as possible
QUAD_DYNAMICS_UPDATE = 0.002 # seconds
CONTROLLER_DYNAMICS_UPDATE = 0.005 # seconds
run = True

def Single_Point2Point():
    # Set goals to go to
    GOALS = [(1,1,2),(1,-1,4),(-1,-1,2),(-1,1,4)]
    YAWS = [0,3.14,-1.54,1.54]
    # Define the quadcopters
    QUADCOPTER={'q1':{'position':[1,0,4],'orientation':[0,0,0],'L':0.3,'r':0.1,'prop_size':[10,4.5],'weight':1.2}}
    # Controller parameters
    CONTROLLER_PARAMETERS = {'Motor_limits':[4000,9000],
                        'Tilt_limits':[-10,10],
                        'Yaw_Control_Limits':[-900,900],
                        'Z_XY_offset':500,
                        'Linear_PID':{'P':[300,300,7000],'I':[0.04,0.04,4.5],'D':[450,450,5000]},
                        'Linear_To_Angular_Scaler':[1,1,0],
                        'Yaw_Rate_Scaler':0.18,
                        'Angular_PID':{'P':[22000,22000,1500],'I':[0,0,1.2],'D':[12000,12000,0]},
                        }
```

```python
# Catch Ctrl+C to stop threadsl
signal.signal(signal.SIGINT, signal_handler)
# Make objects for quadcopter, gui and controller
quad = quadcopter.Quadcopter(QUADCOPTER)
gui_object = gui.GUI(quads=QUADCOPTER)
ctrl = controller.Controller_PID_Point2Point(quad.get_state,quad.get_time,quad.set_motor_speeds,params=CONTROLLER_PARAMETERS,quad_identifier='q1')
# Start the threads
quad.start_thread(dt=QUAD_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)
ctrl.start_thread(update_rate=CONTROLLER_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)
# Update the GUI while switching between destination poitions
while(run==True):
    for goal,y in zip(GOALS,YAWS):
        ctrl.update_target(goal)
        ctrl.update_yaw_target(y)
        for i in range(300):
            gui_object.quads['q1']['position'] = quad.get_position('q1')
            gui_object.quads['q1']['orientation'] = quad.get_orientation('q1')
            gui_object.update()
quad.stop_thread()
ctrl.stop_thread()

Multi_Point2Point():
# Set goals to go to
GOALS_1 = [(-1,-1,4),(1,1,2)]
GOALS_2 = [(1,-1,2),(-1,1,4)]
# Define the quadcopters
QUADCOPTERS={'q1':{'position':[1,0,4],'orientation':[0,0,0],'L':0.3,'r':0.1,'prop_size':[10,4.5],'weight':1.2},
    'q2':{'position':[-1,0,4],'orientation':[0,0,0],'L':0.15,'r':0.05,'prop_size':[6,4.5],'weight':0.7}}
```

```python
# Controller parameters
CONTROLLER_1_PARAMETERS = {'Motor_limits':[4000,9000],
                    'Tilt_limits':[-10,10],
                    'Yaw_Control_Limits':[-900,900],
                    'Z_XY_offset':500,
                    'Linear_PID':{'P':[300,300,7000],'I':[0.04,0.04,4.5],'D':[450,450,5000]},
                    'Linear_To_Angular_Scaler':[1,1,0],
                    'Yaw_Rate_Scaler':0.18,
                    'Angular_PID':{'P':[22000,22000,1500],'I':[0,0,1.2],'D':[12000,12000,0]},
                    }
CONTROLLER_2_PARAMETERS = {'Motor_limits':[4000,9000],
                    'Tilt_limits':[-10,10],
                    'Yaw_Control_Limits':[-900,900],
                    'Z_XY_offset':500,
                    'Linear_PID':{'P':[300,300,7000],'I':[0.04,0.04,4.5],'D':[450,450,5000]},
                    'Linear_To_Angular_Scaler':[1,1,0],
                    'Yaw_Rate_Scaler':0.18,
                    'Angular_PID':{'P':[22000,22000,1500],'I':[0,0,1.2],'D':[12000,12000,0]},
                    }

# Catch Ctrl+C to stop threads
signal.signal(signal.SIGINT, signal_handler)
# Make objects for quadcopter, gui and controllers
gui_object = gui.GUI(quads=QUADCOPTERS)
quad = quadcopter.Quadcopter(quads=QUADCOPTERS)
ctrl1 = controller.Controller_PID_Point2Point(quad.get_state,quad.get_time,quad.set_motor_speeds,params=CONTROLLER_1_PARAMETERS,quad_identifier='q1')
ctrl2 = controller.Controller_PID_Point2Point(quad.get_state,quad.get_time,quad.set_motor_speeds,params=CONTROLLER_2_PARAMETERS,quad_identifier='q2')
# Start the threads
quad.start_thread(dt=QUAD_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)
ctrl1.start_thread(update_rate=CONTROLLER_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)
ctrl2.start_thread(update_rate=CONTROLLER_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)
```

```python
    # Update the GUI while switching between destination poitions
    while(run==True):
        for goal1,goal2 in zip(GOALS_1,GOALS_2):
            ctrl1.update_target(goal1)
            ctrl2.update_target(goal2)
            for i in range(150):
                for key in QUADCOPTERS:
                    gui_object.quads[key]['position'] = quad.get_position(key)
                    gui_object.quads[key]['orientation'] = quad.get_orientation(key)
                gui_object.update()
    quad.stop_thread()
    ctrl1.stop_thread()
    ctrl2.stop_thread()

def Single_Velocity():
    # Set goals to go to
    GOALS = [(0.5,0,2),(0,0.5,2),(-0.5,0,2),(0,-0.5,2)]
    # Define the quadcopters
    QUADCOPTER={'q1':{'position':[0,0,0],'orientation':[0,0,0],'L':0.3,'r':0.1,'prop_size':[10,4.5],'weight':1.2}}
    # Controller parameters
    CONTROLLER_PARAMETERS = {'Motor_limits':[4000,9000],
                        'Tilt_limits':[-10,10],
                        'Yaw_Control_Limits':[-900,900],
                        'Z_XY_offset':500,
                        'Linear_PID':{'P':[2000,2000,7000],'I':[0.25,0.25,4.5],'D':[50,50,5000]},
                        'Linear_To_Angular_Scaler':[1,1,0],
                        'Yaw_Rate_Scaler':0.18,
                        'Angular_PID':{'P':[22000,22000,1500],'I':[0,0,1.2],'D':[12000,12000,0]},
                        }

    # Catch Ctrl+C to stop threads
    signal.signal(signal.SIGINT, signal_handler)
    # Make objects for quadcopter, gui and controller
    quad = quadcopter.Quadcopter(QUADCOPTER)
    gui_object = gui.GUI(quads=QUADCOPTER)
    ctrl = controller.Controller_PID_Velocity(quad.get_state,quad.get_time,quad.set_motor_speeds,params=CONTROLLER_PARAMETERS,quad_identifier='q1')
    # Start the threads
    quad.start_thread(dt=QUAD_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)
    ctrl.start_thread(update_rate=CONTROLLER_DYNAMICS_UPDATE,time_scaling=TIME_SCALING)

    # Update the GUI while switching between destination poitions
    while(run==True):
        for goal in GOALS:
            ctrl.update_target(goal)
            for i in range(150):
                gui_object.quads['q1']['position'] = quad.get_position('q1')
                gui_object.quads['q1']['orientation'] = quad.get_orientation('q1')
                gui_object.update()
    quad.stop_thread()
    ctrl.stop_thread()

def parse_args():
    parser = argparse.ArgumentParser(description="Quadcopter Simulator")
    parser.add_argument("--sim", help='single_p2p, multi_p2p or single_velocity', default='single_p2p')
    parser.add_argument("--time_scale", type=float, default=-1.0, help='Time scaling factor. 0.0:fastest,1.0:realtime,>1:slow, ex: --time_scale 0.1')
    parser.add_argument("--quad_update_time", type=float, default=0.0, help='delta time for quadcopter dynamics update(seconds), ex: --quad_update_time 0.002')
    parser.add_argument("--controller_update_time", type=float, default=0.0, help='delta time for controller update(seconds), ex: --controller_update_time 0.005')
    return parser.parse_args()

def signal_handler(signal, frame):
    global run
    run = False
    print('Stopping')
    sys.exit(0)

if __name__ == "__main__":
    args = parse_args()
    if args.time_scale>=0: TIME_SCALING = args.time_scale
    if args.quad_update_time>0: QUAD_DYNAMICS_UPDATE = args.quad_update_time
    if args.controller_update_time>0: CONTROLLER_DYNAMICS_UPDATE = args.controller_update_time
    if args.sim == 'single_p2p':
        Single_Point2Point()
    elif args.sim == 'multi_p2p':
        Multi_Point2Point()
    elif args.sim == 'single_velocity':
        Single_Velocity()
```

Sources :

https://github.com/abhijitmajumdar/Quadcopter_simulator

http://ardupilot.org/copter/docs/advanced-pixhawk-quadcopter-wiring-chart.html

https://docs.px4.io/en/

https://mscipio.github.io/post/bss-shogun-python/

http://bass-db.gforge.inria.fr/fasst/