FL-AIR Framework User's Guide

- Rafik CHENNOUF
- Sofiane LAMROUS

Juin 2017

Table of Contents

I) Introduction

What is FL-AIR?

FL-AIR is an open source framework written by the Heudiasyc laboratory in C++ language. It aims at helping the creation of application for robots and more specially for UAVs. Many templates are included in the FL-AIR package to help the user to start. It's important to know that FL-AIR has a cross compiler capable of creating two executable files, one to be executed on the UAV target and the other one on the simulator. It is included in the framework, so you can test your program safely before running it on drones.

Why FL-AIR?

Many frameworks of robotics exist so why choosing FL-AIR instead of an other?

Because:

- ✔ FL-AIR is specifically made for UAVs and it allows to develop programs compatible with several types of drone.
- ✔ Cross compiler and simulator, thanks to the flair simulator you will be able to test your programs before running them on real drones.
- ✔ Open source: All the Fl-AIR framework code is open source.
- ✔ Rich library that allows to code more fluently.
- ✔ Program structure that has already been tested and that is reliable.

User guide:

This user guide will teach you from the beginning the important things that you have to know in order to use FL-AIR.
This guide will provide you knowledge and practice so at the end of this course you will be able to write software by yourselves using the framework FL-AIR. This document is not a programming course, it assumes that you've studied C++ in sufficient depth to read, write, and understand code in that language.

NOTE: This manual was written for the 0.0.2 version of Fl-AIR. On May 31, 2017 version 0.1.0 was made available, resulting in some changes on objects but the general principle of the programs was not altered.

II) How to use Fl-AIR ?

This section will explain how to install Fl-AIR on your computer. This part is very important and must be read and understood before starting to work with the framework.

1) Prerequisites

a) Linux

First of all, make sure that you have a Linux distribution on your computer. The recommended distribution is *Mint* downloadable here : https://linuxmint.com/. This distribution is popular, user-friendly and simple to install. However, you are not forced to choose *Mint*, you can also work with another distribution like *Ubuntu* for example. Whatever your choice, be careful to choose the 64-bit version if you have a recent computer.
As Fl-AIR works on Linux, make sure to know how to use this operating system through the terminal where a lot of operations will be executed. For beginners, this tutorial http://ryanstutorials.net/linuxtutorial/ will give you some knowledge about the Bash command line interface. It is important to know that on the target (uav), unlike the computer, there is only a root account without password. Moreover, only two text editors are available on the target : *vi* and *nano*.
Subsequently, for all the tutorial, shell commands that have to be executed on your computer will be indicated as follows, with a dollar symbol $:

```
$ command
```

Shell commands for the target will be indicated as follows, with a sharp symbol # :

```
# command
```

Note that the command just above must be executed from a terminal connected to the target directly via a serial connection or the SSH protocol.

b) Serial console

The serial console is used to communicate with the target (uav) either to access U-Boot or a Linux console. The connection is made through a serial port which is a cheap way to allow software developers to directly access the embedded computer. This is invaluable for debugging. Most chip sets designed for embedded computers have a serial port precisely for this purpose. It is important to know that if the SSH connection does not work, the serial console would be the only way to connect to the target because it does not need a network connection.
If you want more information on the serial console and if you want to know how to access it from Linux, consult the following links :
http://www.tldp.org/HOWTO/Remote-Serial-Console-HOWTO/intro.html

https://www.cyberciti.biz/hardware/5-linux-unix-commands-for-connecting-to-theserial-console/

Regarding U-Boot, it is a primary boot loader used in embedded devices to package the instructions to boot the device's operating system kernel. For further information :
https://wiki.openwrt.org/doc/techref/bootloader/uboot

On Linux, you can use a graphical serial terminal called “CuteCom” :
http://cutecom.sourceforge.net/.

c) SSH connection

Secure Shell (SSH) provides a secure channel over an unsecured network in a client-server architecture, connecting an SSH client application (your computer) with an SSH server (the target = the drone = the uav).

To connect to an SSH server, you must enter the following command :

```
$ ssh user@server
```

user is the user name used to connect to the server while *server* is either the server's name or its IP address.

For example, to connect to a server named *storage* as the user *toto* :

```
$ ssh toto@storage
```

To connect the target with its IP address :

```
$ ssh toto@192.168.3.6
```

It should be noted that the first time you connect to an SSH server, it will be added to the known hosts file of your computer.

You can find a more detailed SSH tutorial here :
https://support.suso.com/supki/SSH_Tutorial_for_Linux

d) SCP

Secure copy or SCP is a means of securely transferring computer files between a local host and a remote host or between two remote hosts. It is based on the Secure Shell (SSH) protocol.

To use it, you have to enter the following command :

```
$ scp source_path user@server:/destination_path
```

source_path is the path of the file you want to transfer, *user* is the user name used to connect to the *server* and *destination_path* is the destination path of your file in the server. In practice, *user* is the same as for the SSH connection.

For example, to copy the file *test* from your home directory to the server (IP : *192.168.6.1*) into the folder */home/root* :

```
$ scp ~/test root@192.168.6.1:/home/root
```

For further information : http://www.hypexr.org/linux_scp_help.php

e) SVN

Apache Subversion (SVN) is a software versioning and revision control. SVN is used for the Fl-AIR Framework and works like Git through the terminal.

To check out a working copy from a repository, you must enter :

```
$ svn checkout
```

To update your working copy and bring changes from the repository into your working copy :

```
$ svn update
```

To send changes from your working copy to the repository :

```
$ svn commit
```

If you wish you could also use a visual client such as http://rapidsvn.tigris.org/.

For more details about SVN, follow this link :
https://www.tutorialspoint.com/svn/index.htm.

f) CMake

CMake is cross-platform free and open-source software for managing the build process of software using a compiler-independent method. It supports directory hierarchies and applications that depend on multiple libraries.

CMake is used to automatically generate standard building files such as Makefiles or projects for different IDE, through a single configuration file named *CmakeLists.txt* regardless of the IDE used. Moreover, CMake is able to manage different operating systems.

CMake must be used from the command line. Enter the following command to get the different options :

```
$ cmake
```

For example, to create a project for CodeBlocks IDE, you must execute the next command from a repertory containing a *CmakeLists.txt* file :

```
$ cmake -G "CodeBlocks - Unix Makefiles"
```

For further information, follow this link :
https://www.jetbrains.com/help/clion/2016.3/quick-cmake-tutorial.html

2) Setup your computer

In this part we will see how to proceed to have all the tools needed to start using FLAIR. For example, how to set your work environment or how to install an IDE.

a) Work directory and path variable

To work comfortably it is advisable to create a unique directory that contains all FLAIR source code and libraries.

This command allows you to create this directory located in HOME directory.

```
$ mkdir $HOME/flair
```

Then you can add an environment variable to your *.bashrc* with the path of this directory. It's like creating a shortcut called *$FLAIR_ROOT*, and we will use it to simplify the way how we access to FLAIR main directory.

To do that we have to edit the *.bashrc* file to declare our environment variable.

```
$ nano ~/.bashrc
```

Then add these lines at the end.

```
# variable for Fl-AIR
export FLAIR_ROOT=$HOME/flair
```

CTRL+X to close the file and answer yes to save the modifications (if you used nano to edit the file).
Now we need to reload the *.bashrc* script to create our variable *$FLAIR_ROOT*.

Here is the command :

```
$ source ~/.bashrc
$ echo $FLAIR_ROOT
```

The second line is used to check if the variable has been created.
It's important to know that all Fl-AIR documentation, scripts and *CMakeLists.txt* will use this variable. So it's really recommended that you do not skip this step.

b) IDE installation

In this part we will install an IDE, you can choose between Code::Blocks and Eclipse.

To help you make your decision you have to know that all Fl-AIR documentation is based on Code::Blocks but Eclipse is appropriate as well.

If you have opted for Code::Blocks here is the installation command:

```
$ sudo apt-get install codeblocks codeblocks-contrib
```

If you prefer Eclipse:

```
$ sudo apt-get install eclipse-cdt
```

The environment variable *$FLAIR_ROOT* that we have created in the previous section must be added manually in Code::Blocks because when the IDE is not launched from command line, it does not read environment variables set in *.bashrc*.

Here is how we add a new environment variable on Code::Blocks:

Go to settings/environment menu. Choose environment variables section on the left.
Click on Add and fill the form :
Key: *FLAIR_ROOT*
Value: *path to the FLAIR_ROOT directory*

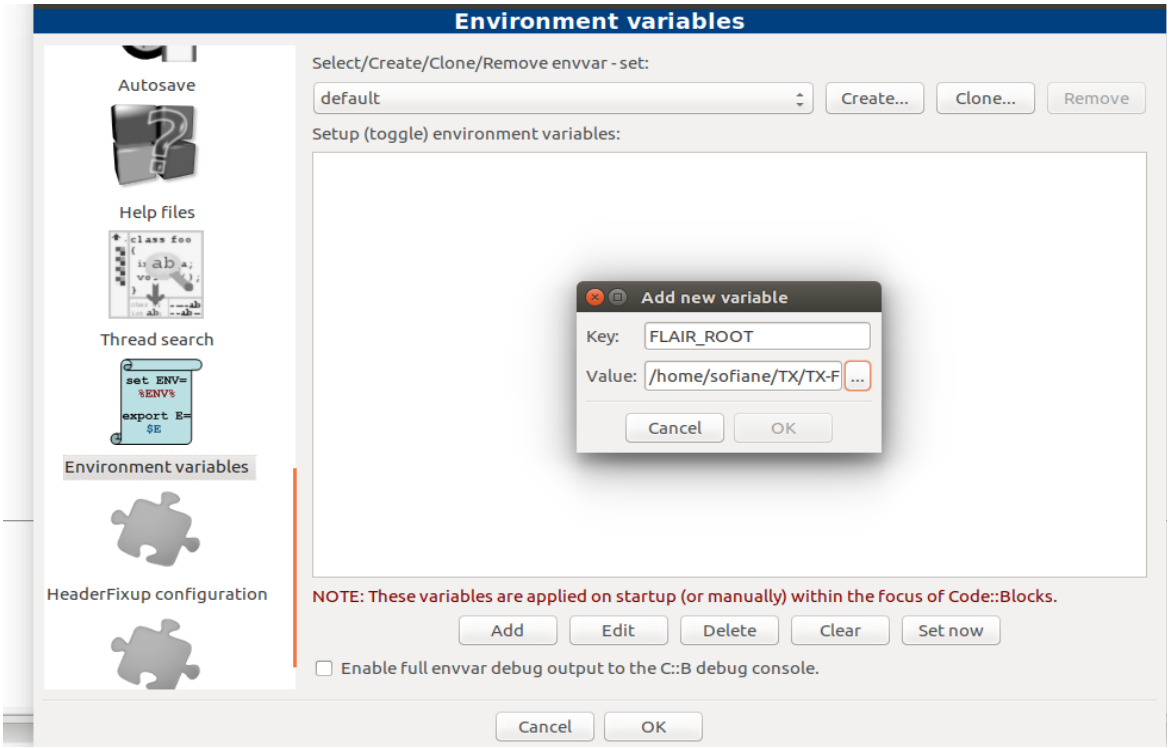


Illustration 1: Add a new environment variable in CodeBlocks

c) Real time process scheduling:

If you are using FLAIR without Xenomai (real-time development framework) you have to modify the configuration file *limits.conf* in order to remove the limit that affect the maximum real time priority allowed for non-privileged processes.
We will use nano for that:

```
$ sudo nano /etc/security/limits.conf
```

And add these lines at the end (your_user is your user login; to get it you can open a terminal and type *whoami*):

```
your_user               soft    rtprio          99
your_user               hard    rtprio          99
```

Let’s take a look at these lines:
The syntax is as follows:

<user><type><item><value>

The type can take three values soft, hard or “–“ :
“soft limit": the values specified with this token can be thought of as default values, for normal system usage.
"hard limit": used for enforcing hard resource limits. The user cannot raise his requirement of system resources above such values.
The “–“ is for enforcing both soft and hard resource limits together.

The *rtprio* is an item used for controlling real time process scheduling, here is an article explaining in detail the real time process scheduling:
https://wiki.archlinux.org/index.php/realtime_process_management.

In order to apply the *limits.conf* changes you have to logout and login of your Linux session.

3) Install Toolchain

a) Cross-toolchain and toolchain

A cross compiler is used to compile your programs and make one version runnable in the flair simulator and a second version for the target UAV.

If you want to learn more about toolchain and the architecture behind here is a link that can help you: http://infocenter.arm.com/help/.

The version that is executable on the target is made by the cross-toolchain, to install the cross compiler copy and paste these lines:

```
$ cd ~
$ wget https://uav.hds.utc.fr/src/toolchain/x86_64-meta-toolchain-flair-arm.sh
$ chmod +x x86_64-meta-toolchain-flair-arm.sh
$ sudo ./x86_64-meta-toolchain-flair-arm.sh
$ rm x86_64-meta-toolchain-flair-arm.sh
```

Here is the command to install the toolchain that will generate an executable for the simulator on your computer.

```
$ cd ~
$ wget https://uav.hds.utc.fr/src/toolchain/x86_64-meta-toolchain-flair-x86_64.sh
$ chmod +x x86_64-meta-toolchain-flair-x86_64.sh
$ sudo ./x86_64-meta-toolchain-flair-x86_64.sh
$ rm x86_64-meta-toolchain-flair-x86_64.sh
```

Environment variable

The toolchain installation scripts generate environment variables but in order to use them you have to reload your *.bashrc* file with this command.

```
$ source ~/.bashr
```

b) Graphics drivers

To use the FLAIR simulator you need to update your graphics drivers and copy them in the library toolchain directory robomap3:

For Nvidia graphics cards

```
$ sudo cp -r /usr/lib/nvidia-340/* /opt/robomap3/1.7.3/core2-64/sysroots/core2-64-poky-linux/usr/lib
```

Here "nvidia-340" is the driver version it could be different you should take the latest version, check this here http://www.nvidia.fr/Download/driverResults.aspx/77575/fr.

For AMD graphics cards

```
$ sudo cp -r /usr/lib/fglrx/* /opt/robomap3/1.7.3/core2-64/sysroots/core2-64-poky-linux/usr/lib
```

Fglrx is the default directory that contains AMD graphics driver.
Please note that ubuntu 16.04 and higher don't support any more fglrx drivers.

For Intel graphics cards :

It's a little bit more complicated, due to several versions and path for the libraries.

The idea is that we're using the system video card driver since it's not included in FL-AIR. The driver itself (eg: i965_dri.so) should be loaded automatically (from /usr/lib/x86_64-linux-gnu/dri/).

To debug this you should launch your 3D graphic program (e.g.: simulator) with the LIBGL_DEBUG environment variable set to verbose.

```
$ cd $FLAIR_ROOT/flair-src/demos/CircleFollower/simulator/build/bin
```

```
$ LIBGL_DEBUG=verbose ./simulator_x4.sh
```

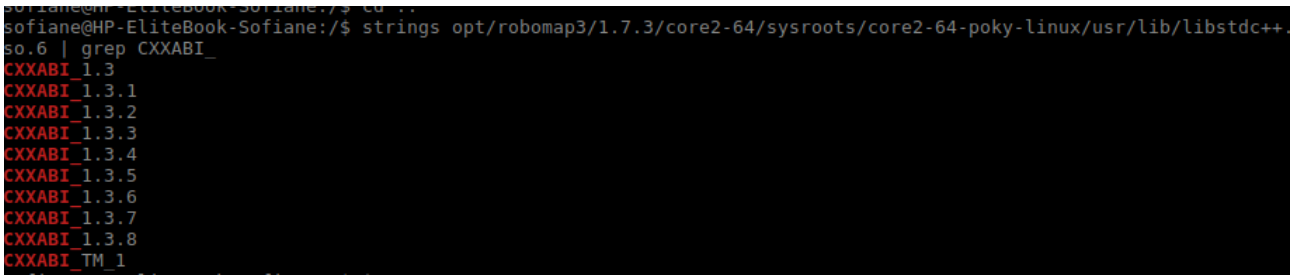
You will probably get an error in this form

```
libGL: dlopen /usr/lib/x86_64-linux-gnu/dri/i965_dri.so failed
(/opt/robomap3/1.7.3/core2-64/sysroots/core2-64-poky-
linux/usr/lib/libstdc++.so.6: version `CXXABI_1.3.9' not found (required
by /usr/lib/x86_64-linux-gnu/dri/i965_dri.so))
```

If you get an error like this it's because the driver doesn't successfully load and may require some specific library versions that are not included in *robomap3* (a FL-AIR directory) . For example in this case, the version of **libstdc++** included in *robomap3* doesn't define the symbol `**CXXABI_1.3.9**' . You can make sure of this missing by searching **CXXABI** in the file libstdc++.so.6 contained in *robomap3* directory.

```
$ strings opt/robomap3/1.7.3/core2-64/sysroots/core2-64-poky-
linux/usr/lib/libstdc++.so.6 | grep CXXABI_
```

It will check if libstdc++.so.6 doesn't really have GXXABI as pretended by the error message. You should get some thing like this, `**CXXABI_1.3.9**' is really missing :



To fix this error we need to preload the libstdc++ library of the system in order to force FL-AIR to use this library instead of the robomap3 one. To achieve this we need to set the LD_PRELOAD environment variable before launching the simulator :

```
$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6
./simulator_x4.sh
```

Stopping here may be enough, but you can get other errors like:

```
i965_dri.so: undefined symbol: nouveau_drm_new
```

As previously you have to preload missing libraries until no more errors.

For laptops with two graphics cards:

If your laptop have two graphics cards it is more complicated because you should disable one of them.

Nvidia and Intel:

You need to have both Nvidia proprietary driver and *nvidia-prime* to get it working. This can be done by running:

```
sudo apt-get install nvidia-340 nvidia-prime
```

Nvidia-340 is the proprietary driver, it can be different depending your graphic card model, you should take the latest version, check this here: http://www.nvidia.fr/Download/driverResults.aspx/77575/fr.

After the installation you will have Nvidia X Server Settings in your dash menu. Open the application, then find PRIME profiles and check NVIDIA as below.

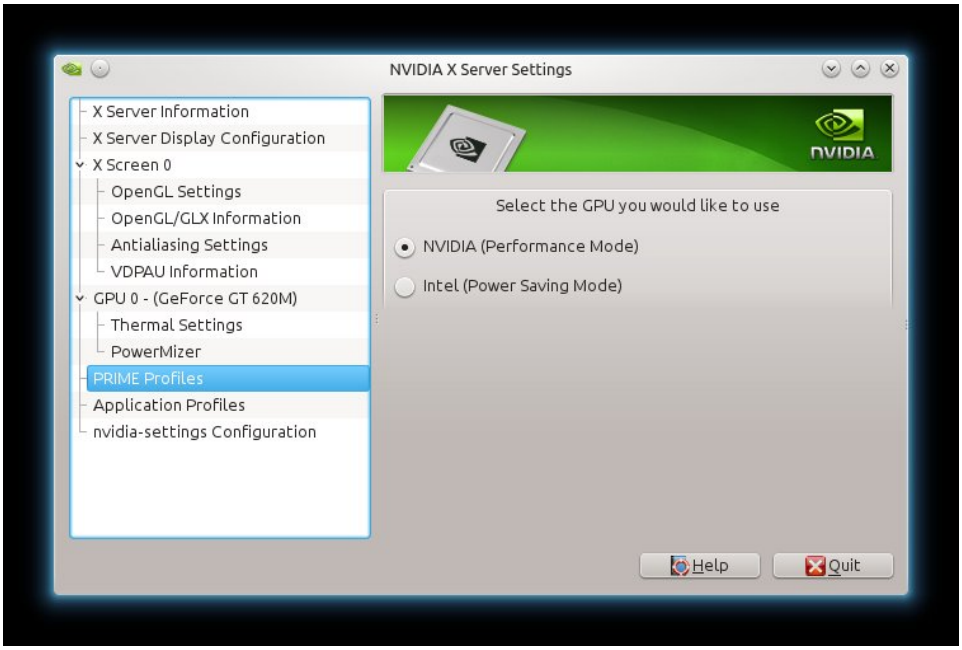


Illustration 2: Nvidia X Server Settings configuration

4) FLAIR SVN repository

a) What is an SVN repository ?

An SVN repository is a repository used for managing any collection of files that are changed or modified over time. It is a collection of files and directories, bundled together in a special database that also records a complete history of all the changes that have ever been made to these files.

More information: https://www.projecthut.com/what-is-svn-repository/

b) How FLAIR SVN is organized ?

There are 3 main root directories:

- flair-src
- flair-dev
- flair-bin

flair-src

This repository contains the Fl-AIR source code.

It is organized as follows:

- demos: Fl-AIR demos are good starting points if you want to write your own application . There is also Skeletons that are also good to begin as they are very minimalistic (these programs will be presented later in this guide).
- lib: FL-AIR libraries sources which contain all source code like the objects that we will use in our code.
- tools: FL-AIR tools sources (ground control station, etc.).

flair-dev

This repository contains all necessary files to develop applications with Fl-AIR.

It is organized as follows:

- cmake-modules: modules for cmake (used for managing the build process of software).
- doc: documentation in HTML (doxygen).
- include: Fl-AIR header files.
- scripts: scripts used to develop with Fl-AIR: for example *flair_compile_all.sh* is a script used to compile all Fl-AIR code.

flair-bin

This repository contains Fl-AIR binaries.

It is organized as follows:

- models: models used by Fl-AIR simulator.
- lib: Fl-AIR compiled libraries.
- tools: Fl-AIR compiled tools like : dualshock3 to connect your PS3 controller or *flairgcs* for the ground station, these tools will be explained later in this guide.

flair-hds repository

Some additional software is available for Heudiasyc users only. This software is not included in flair repositories because of license incompatibility with flair.

flair-hds repository is organised as flair-dev, flair-bin and flair-src repositories. It contains the following directories:

- src: the sources
- bin: all necessary files to develop applications with flair-hds libs
- dev: binaries

Each root directory forms an SVN repository which is organized as below:

```
Root
 | -> tags
    | -> latest
    | -> 1.0
    | -> 1.1
    | -> 2.0
    | -> ...
 | -> trunk
 | -> branches
    | -> testing
    | -> ...
```

- Tags

The tags are the stable versions of the repository. As these versions are stable you can't modify them and they are supposed to be functional. The latest tag is the latest stable version, we will use this one. Note that this guide was written in march 2017 and all the explanations are relevant for the version 0.0.2.

- Trunk

The trunk branch is the version still under development.

- Other branches

The other branches are used to order other development versions that we can modify without altering the trunk version.

c) Get repositories

Depending on what you are doing with Fl-AIR, there are several options to get the repositories.

Only developing applications based on latest Fl-AIR version

To get the latest version of Fl-AIR run the following commands:

```
$ cd $FLAIR_ROOT
$ svn co https://devel.hds.utc.fr/svn/flair-dev/tags/latest flair-dev
$ svn co https://devel.hds.utc.fr/svn/flair-bin/tags/latest flair-bin
$ svn co https://devel.hds.utc.fr/svn/flair-src/tags/latest flair-src
```

Developing applications based on multiple versions of Fl-AIR

If you need more than one version of Fl-AIR on your computer, one solution is to do the following:

```
$ cd $FLAIR_ROOT
$ svn co https://devel.hds.utc.fr/svn/flair-dev flair-dev_svn
$ svn co https://devel.hds.utc.fr/svn/flair-bin flair-bin_svn
$ svn co https://devel.hds.utc.fr/svn/flair-src flair-src_svn
```

These commands will sync all the repositories. If you want to sync only some directories you can use *svn update --set-depth exclude "folderName"* command to remove this folder from your working directory:

To work on a FL-AIR version you can make a symbolic link to this version that you need.

```
$ cd $FLAIR_ROOT
$ ln -s flair-dev_svn/tags/0.0.1 flair-dev
$ ln -s flair-bin_svn/tags/0.0.1 flair-bin
$ ln -s flair-src_svn/tags/0.0.1 flair-src
```

Here we have chosen the 0.0.1 tag version.

d) Updating repositories

To update your local repositories:

```
$ svn up $FLAIR_ROOT/flair-dev --accept theirs-full
$ svn up $FLAIR_ROOT/flair-bin --accept theirs-full
$ svn up $FLAIR_ROOT/flair-src
```

In some cases, you can have a conflict between local file and remote files. This can happen if you recompiled all FL-AIR. In this case, and if you are sure you can discard your changes, answer "*tf* "to the conflict question.

e) Compile libraries, tools and demos

Compiling all is not really necessary, as everything necessary for development is already compiled in *flair_bin* repository. But it is needed if you did not installed the toolchains in their defaults directories.

For developers of the Framework, it is a good way to check that everything is compiling like all the libraries and their own programs.

To compile all Fl-AIR code, you can execute the dedicated script:

```
$ $FLAIR_ROOT/flair-dev/scripts/flair_compile_all.sh
```

If it is the first time you execute this script, you must answer yes to the question Compile all from scratch. This will create every projects using *cmake*.

III) Your first program

In this section, we will learn how to develop and compile your first program and execute it on the simulator. We will simulate a drone following a square trajectory around a character able to move inside a motion capture room. The version of Fl-AIR used here is **0.0.2**.

a) Where to start ?

First of all, we will create a working directory called *SquareFollower* into *$FLAIR_ROOT/flair-src/demos.* Every programs that are not directly part of the main Fl-AIR library must be written in the directory *demos.* However if you want to add some features to the framework, you need to write them into *$FLAIR_ROOT/flair-src/lib.*

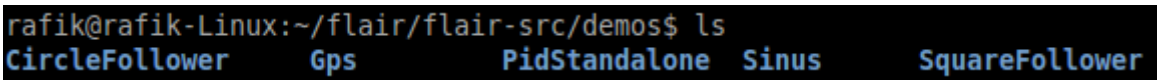


Illustration 3 : "demos" directory

As a *CircleFollower* program already exists, we will use it as an example because it is based on the same principle. Like the *CircleFollower* directory, we will create two directories, one called *simulator* and other *uav*. The simulator directory will contain the source files used to display the simulator and the uav will contain the source files of our square trajectory. Both directories must contain a directory called *src* where you will put your source files and a CMake file named *CmakeLists.txt* used to generate your project files. If you don't know how to write a CMake file, refer you to the tutorial provided in the first chapter and help you with existing files that you can find in others demos.

Let's go to the *simulator* directory.

b) Simulator directory

Inside the directory *src*, you will find a *main.cpp* file which contains a source code used to launch the simulator. We are going to briefly explain how this file is written because its is approximately the same file for all applications. However, if you want to know concretely how the simulator is made, refer to the simulator source code in *$FLAIR_ROOT/flair-src/lib/FlairSimulator* and *$FLAIR_ROOT/flair-src/tools/FlairGCS*.

First of all, make sure the simulator is running. Move to the following folder that should contain your *src* directory and your *CmakeLists.txt* file.

```
$ cd $FLAIR_ROOT/flair-src/demos/SquareFollower/simulator
```

We are going to compile our program for the target and for the computer with the script *$FLAIR_ROOT/flair-dev/scripts/cmake_codeblocks.sh* that will generate a CodeBlocks project. Remember that CMake also manages cross-compiling. Thus, two project directories will be created, *build_x86_64* for the computer and *build_arm* for the target which has an ARM architecture. If you want to know more about ARM, follow this link : https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm/home.html

To create both project directories :

```
$ $FLAIR_ROOT/flair-dev/scripts/cmake_codeblocks.sh
```

A symbolic link called *build* that refers to the repertory *build_x86_64* will also be created.

Now that our project has been created, we need to compile it for our computer. We will see how to compile it for the target later.

```
$ cd build
$ make
```

The *make* command is used to compile your program, more information about it : http://www.computerhope.com/unix/umake.htm

Two generated executable files will be found in *bin* : *SquareFollower_simulator_rt* and *SquareFollower_simulator_nrt*.

Why two executable files instead of one? If you go back to *$FLAIR_ROOT/flair-src/demos/SquareFollower/simulator* and open your *CmakeLists.txt* file, you should see at the end of the file:

```
#real time executable
ADD_EXECUTABLE(${PROJECT_NAME}_rt
  ${SRC_FILES}
)

TARGET_LINK_LIBRARIES(${PROJECT_NAME}_rt ${FLAIR_LIBRARIES_RT})

#non real time executable
ADD_EXECUTABLE(${PROJECT_NAME}_nrt
  ${SRC_FILES}
)

TARGET_LINK_LIBRARIES(${PROJECT_NAME}_nrt ${FLAIR_LIBRARIES_NRT})
```

Illustration 4 : CMakeLists.txt

You see that your file is configured to generate a real and non real time executables (cross-compilation). If you require more information about the real time concept, follow this link : https://en.wikipedia.org/wiki/Real-time_computing

Now go back to *$FLAIR_ROOT/flair-src/demos/SquareFollower/simulator/build/bin.* You should see a bash script called *simulator_x4.sh* :

```
1  #! /bin/bash
2
3  . $FLAIR_ROOT/flair-dev/scripts/ubuntu_cgroup_hack.sh
4
5  if [ -f /proc/xenomai/version ];then
6    EXEC=./SquareFollower_simulator_rt
7  else
8    EXEC=./SquareFollower_simulator_nrt
9  fi
10
11 $EXEC -n x4_0 -t x4 -p 9000 -a 127.0.0.1 -x setup_x4.xml
12       -o 10 -m $FLAIR_ROOT/flair-bin/models
13       -s $FLAIR_ROOT/flair-bin/models/indoor_flight_arena.xml
```

Illustration 5 : simulator_x4.sh

This script is used to run your program properly.
At line 3, a script is used to bypass the *cgroup* restriction. Control groups (*cgroups*) are a kernel mechanism for grouping, tracking, and limiting the resource usage of tasks.

From line 5 to 9, we check if your computer has the real time framework Xenomai. If it is installed, the real time program will be launched, otherwise, it will be the non real time program.
More information about Xenomai here : https://en.wikipedia.org/wiki/Xenomai.

Finally, from line 11 to 13, we run our program with a number of arguments. All these parameters are required in the *main.cpp* file in *$FLAIR_ROOT/flair-src/demos/SquareFollower/simulator/src*. Open this file, you will find a function called *parseOptions* :

```
41  void parseOptions(int argc, char** argv) {
42    try {
43          CmdLine cmd("Command description message", ' ', "0.1");
44
45          ValueArg<string> nameArg("n","name","uav name, also used for vrpn",true,"x4","string");
46          cmd.add( nameArg );
47
48          ValueArg<string> xmlArg("x","xml","xml file",true,"./reglages.xml","string");
49          cmd.add( xmlArg );
50
51          ValueArg<int> portArg("p","port","ground station port",true,9002,"int");
52          cmd.add( portArg );
53
54          ValueArg<string> addressArg("a","address","ground station address",true,"127.0.0.1","string");
55          cmd.add( addressArg );
56
57          ValueArg<string> typeArg("t","type","uav type, x4 or x8",true,"x4","string");
58          cmd.add( typeArg );
59
60          ValueArg<int> optiArg("o","opti","optitrack time ms",false,0,"int");
61          cmd.add( optiArg );
62
63  #ifdef GL
64          ValueArg<string> mediaArg("m","media","path to media files",true,"./","string");
65          cmd.add( mediaArg );
66
67          ValueArg<string> sceneArg("s","scene","path to scene file",true,"./voliere.xml","string");
68          cmd.add( sceneArg );
69  #endif
70
71          cmd.parse( argc, argv );
```

Illustration 6 : main.cpp

All the arguments required to launch the simulator program are defined in that function :

- -n : name of your uav, you can use any name. Ex : *x4_0*
- -t : type of your uav, 4 or 8 propellers. Ex : *x4*
- -p : port that will be used by the ground station program. Ex : *9000*
- -a : address of the ground station = local address of your computer = localhost = *127.0.0.1*
- -x : xml file to configure your GUI (simulator), it must be written in *bin*. Ex : *setup_x4.xml*
- -o : optitrack acquisition period in ms. Ex : *10*
- -m : path to media files, that is to say the files used for the simulator scenery. Ex : *$FLAIR_ROOT/flair-bin/models*
- -s : path to the simulator main scene file. Ex : *$FLAIR_ROOT/flair-bin/models/indoor_flight_arena.xml*

It is also recommended to read and understand the *main* function.

Now that you understand how the simulator is set up, let's try it.
First, run the ground control station in one terminal :

```
$ $FLAIR_ROOT/flair-bin/tools/scripts/launch_flairgcs.sh
```

Then, run the simulator in a second terminal :

```
$ cd $FLAIR_ROOT/flair-src/demos/SquareFollower/simulator/build/bin
$ ./simulator_x4.sh
```

Now, let's go to the *uav* directory.

c) Uav directory

The tree of this directory is the same as the simulator directory. Therefore, we will only focus on the source code you can find in *$FLAIR_ROOT/flair-src/demos/SquareFollower/uav/src.* If you open the *main.cpp* file, you will find the function *parseOptions* we introduced before. This function is not very different from the previous one, although there are two new arguments :

- -l : log repository. Ex : */tmp*
- -d : dualshock3 controller port. Ex : *20 000*

We will see later how to use a dualshock3 to control the drone.

```
#! /bin/bash

. $FLAIR_ROOT/flair-dev/scripts/ubuntu_cgroup_hack.sh

if [ -f /proc/xenomai/version ];then
  EXEC=./SquareFollower_rt
else
  EXEC=./SquareFollower_nrt
fi

$EXEC -n x4_0 -a 127.0.0.1 -p 9000 -l /tmp -x setup_x4.xml -t x4_simu
```

Illustration 7 : x4.sh

All these arguments are provided in the following launch script : *$FLAIR_ROOT/flair-src/demos/SquareFollower/uav/build/bin/x4.sh.*

To test the program, you need first to launch the ground control station in one terminal and the simulator in a second terminal as seen previously. Then in a third terminal, you run *x4.sh* :

```
$ cd $FLAIR_ROOT/flair-src/demos/SquareFollower/uav/build/bin
$ ./x4.sh
```

In the *main.cpp* file, don't forget to also read and understand the *main* function.

Now, let's check the *SquareFollower.h* and *SquareFollower.cpp* files.

d) SquareFollower implementation

Here we are going to explain what are the functions used in *SquareFollower.cpp* and how they work. Note that the *SquareFollower* class must inherit from the *UavStateMachine* class. Indeed, all the possible states of the drone are represented by a state machine. More information can be found here : https://en.wikipedia.org/wiki/Finite-state_machine.
In fact, all the classes within your applications will have to be child of the *UavStateMachine* class.

1) Constructor

The constructor *SquareFollower* is used to configure the VRPN and the layout of the ground control station. For information, the Virtual-Reality Peripheral Network (VRPN) is a set of classes within a library and a set of servers that are designed to implement a network-transparent interface between application programs and the set of physical devices (tracker, etc.) used in a virtual-reality (VR) system. The idea is to have a PC or other host at each VR station that controls the peripherals (tracker, analog, button, etc.). VRPN provides connections between the application and all of the devices using the appropriate class-of-service for each type of device sharing this link. The application remains unaware of the network topology. In our case, the physical devices used as trackers are part of a larger motion capture system called OptiTrack. More information about it will be given later in the tutorial. In the meantime, you can find further information here :
http://www.vrgeeks.org/vrpn/tutorial---use-vrpn
http://optitrack.com/motion-capture-robotics/

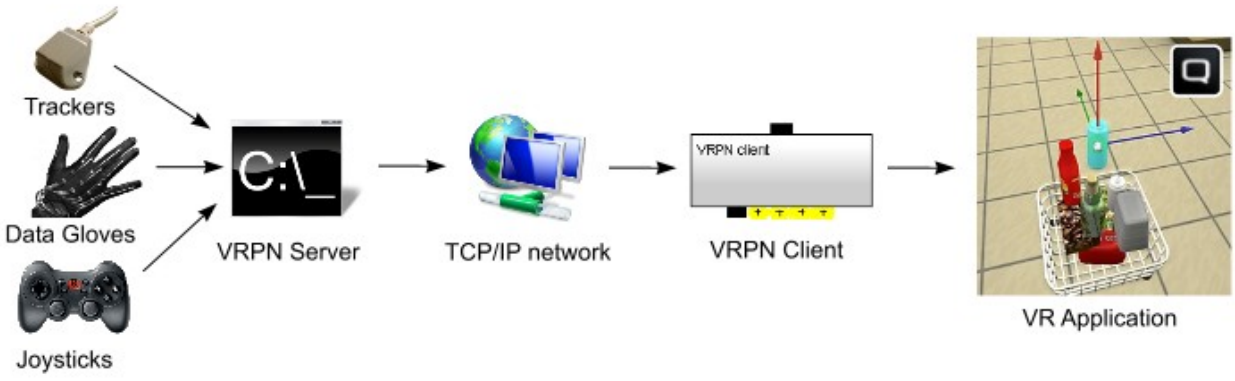


Illustration 8 : How the VRPN works

Here, the VR Application referred to our simulator. Let's back to the constructor.

```
42      uav->SetupVRPNAutoIP(uav->ObjectName());
43
44      startSquare=new PushButton(GetButtonsLayout()->NewRow(),"start_square");
45      stopSquare=new PushButton(GetButtonsLayout()->LastRowLastCol(),"stop_square");
46
47      if(uav->GetVrpnClient()->UseXbee()==true) {
48          targetVrpn=new MetaVrpnObject(uav->GetVrpnClient(),"target",1);
49      } else {
50          targetVrpn=new MetaVrpnObject(uav->GetVrpnClient(),"target");
51      }
52
53      getFrameworkManager()->AddDeviceToLog(targetVrpn);
```

Illustration 9 : SquareFollower.cpp (Constructor)

In line 42, we are initializing the network server (VRPN) that will be used to stream the drone data.
From line 44 to 45, we are adding push buttons to the ground control station. These buttons will be used to start and stop the square.
From line 47 to 51, we are creating a VRPN object from the VPRN server created for our drone. Our drone will be represented by a VRPN object. Here the method *UseXbee()* referred to XBee modules. More information here : https://en.wikipedia.org/wiki/Xbee.
Line 53 is used to configure logs with our VRPN object (drone). The added object will be automatically logged once logging is started.

```
55      square=new TrajectoryGenerator2DSquare(uav->GetVrpnClient()->GetLayout()->NewRow(),"square");
56      uav->GetVrpnObject()->xPlot()->AddCurve(square->Matrix()->Element(0,0),DataPlot::Blue);
57      uav->GetVrpnObject()->yPlot()->AddCurve(square->Matrix()->Element(0,1),DataPlot::Blue);
58      uav->GetVrpnObject()->VxPlot()->AddCurve(square->Matrix()->Element(1,0),DataPlot::Blue);
59      uav->GetVrpnObject()->VyPlot()->AddCurve(square->Matrix()->Element(1,1),DataPlot::Blue);
60      uav->GetVrpnObject()->XyPlot()->AddCurve(square->Matrix()->Element(0,1),square->Matrix()->Element(0,0),DataPlot::Blue,"square");
61
62      uX=new Pid(setupLawTab->At(1,0),"u_x");
63      uX->UseDefaultPlot(graphLawTab->NewRow());
64      uY=new Pid(setupLawTab->At(1,1),"u_y");
65      uY->UseDefaultPlot(graphLawTab->LastRowLastCol());
```

Illustration 10 : SquareFollower.cpp (Constructor)

In line 55, a *TrajectoryGenerator2DSquare* object is created. This class will be discussed later and it is used to concretely implement a square trajectory. Generally, when you want to implement a new trajectory, you use a different class and a file separated from your main class file.

From line 56 to 60, we are adding some graphics to our ground control station to represent the positions x and y as a function of time, the velocities vx and vy as a function of time, the position x as a function of y. All these graphics will be available in the simulator *vrpn* tab.
Also, it is important to know that position and velocity are represented through the following matrix :

$$\begin{pmatrix} x & y \\ vx & vy \end{pmatrix} = \begin{pmatrix} x & y \\ \dot{x} & \dot{y} \end{pmatrix}$$

From line 62 to 65, we are adding control laws (PID for x and y) and graphics of theses laws to our ground control station. Everything will be available in the simulator *control laws* tab.

```
67      customReferenceOrientation= new AhrsData(this,"reference");
68      uav->GetAhrs()->AddPlot(customReferenceOrientation,DataPlot::Yellow);
69      AddDataToControlLawLog(customReferenceOrientation);
70
71      customOrientation=new AhrsData(this,"orientation");
```

Illustration 11 : SquareFollower.cpp (Constructor)

The lines 67 to 71 are used to define a custom Attitude and heading reference system (AHRS). An AHRS consists of sensors on three axes that provide attitude information for aircraft, including roll, pitch and yaw. The class *AhrsData* will provide further measurements such as quaternion, rotational angles values and data collected from the IMU's sensors (gyroscope, accelerometer, magnetometer). These data will be available in the simulator *imu* tab. More information about AHRS in the following method.

2) GetOrientation

The method *GetOrientation* is used to get the drone orientation within an AHRS class. This method is inherited from the *UavStateMachine* class.

```
78      //get yaw from vrpn
79      Euler vrpnEuler;
80      GetUav()->GetVrpnObject()->GetEuler(vrpnEuler);
81
82      //get roll, pitch and w from imu
83      Quaternion ahrsQuaternion;
84      Vector3D ahrsAngularSpeed;
85      GetDefaultOrientation()->GetQuaternionAndAngularRates(ahrsQuaternion, ahrsAngularSpeed);
```

Illustration 12 : SquareFollower.cpp (GetOrientation)

Lines 79 and 80 are used to get Euler angles (yaw rotation) of the drone from the VRPN. From line 83 to 85, we are getting the roll, the pitch and the angular speed (w) from the IMU's sensors in the form of a quaternion. Using quaternions instead of Euler angles avoid a phenomenon called gimbal lock which can result when, for example in pitch/yaw/roll rotational systems, the pitch is rotated 90° up or down, so that yaw and roll then correspond to the same motion, and a degree of freedom of rotation is lost which can be very dangerous especially when the aircraft is in a steep dive or ascent.
More information about quaternions and spatial rotation here :
https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
Information about Euler angles here : https://en.wikipedia.org/wiki/Euler_angles

```
87      Euler ahrsEuler=ahrsQuaternion.ToEuler();
88      ahrsEuler.yaw=vrpnEuler.yaw;
89      Quaternion mixQuaternion=ahrsEuler.ToQuaternion();
90
91      customOrientation->SetQuaternionAndAngularRates(mixQuaternion,ahrsAngularSpeed);
92
93      return customOrientation;
```

Illustration 13 : SquareFollower.cpp (GetOrientation)

From line 87 to 89, we convert the previously obtained quaternion into Euler angles.
Then we set the yaw angle to the previous yaw value extracted from the VRPN.
Finally we create a new quaternion with the previously added yaw value that we return within an AHRS class.

Note here that it is not possible to get an accurate value of the yaw angle directly from the IMU's sensors because of a magnetic field generated by the motors which distorts the value given by the magnetometer. That's why we have to extract the yaw from the VRPN and then insert it in the quaternion.

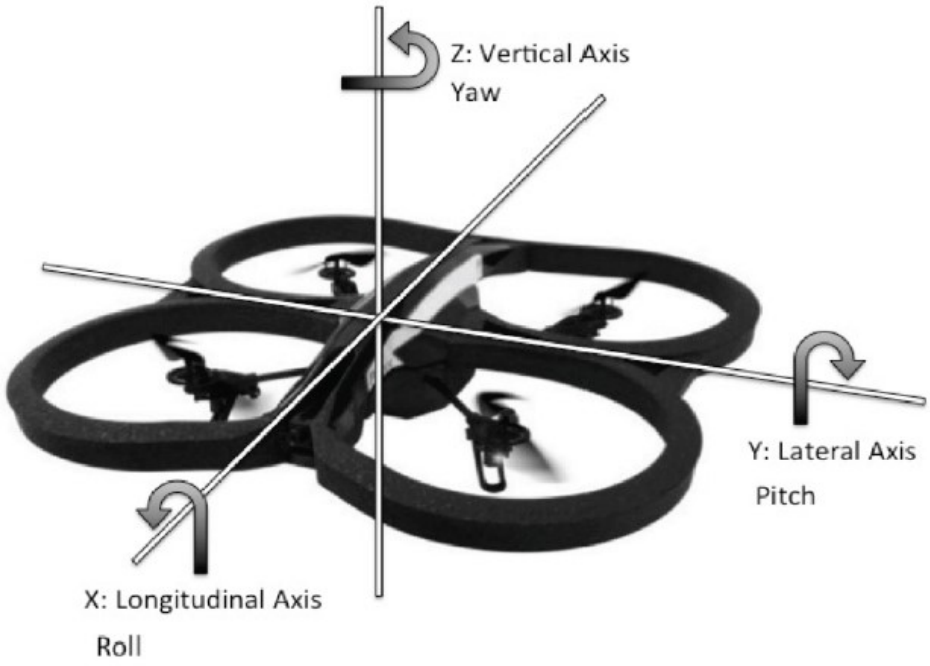


Illustration 14 : Yaw, Roll and Pitch in a Parrot AR Drone

http://forums.ni.com/legacyfs/online/196640_F2.jpg

3) AltitudeValues

The method *AltitudeValues* returns the position and the velocity on the z axis.

```
97      Vector3D uav_pos,uav_vel;
98
99      GetUav()->GetVrpnObject()->GetPosition(uav_pos);
100     GetUav()->GetVrpnObject()->GetSpeed(uav_vel);
101     //z and dz must be in uav's frame
102     z=-uav_pos.z;
103     dz=-uav_vel.z;
```

Illustration 15 : SquareFollower.cpp (method AltitudeValues)

In lines 99 and 100, we get the position and the velocity from the VRPN then in lines 102 and 103, we extract the position and the velocity on the z axis. As we are working in the uav's coordinate system, we must take the additive inverse (-) because z goes down.

Illustration 16 : Uav's coordinate system

4) VrpnPositionHold

This method is used when the drone is no longer in motion (holding) to get its position on the x and y axis as well as its yaw angle.

```
256     Euler vrpn_euler;
257     Vector3D vrpn_pos;
258
259     GetUav()->GetVrpnObject()->GetEuler(vrpn_euler);
260     yawHold=vrpn_euler.yaw;
261
262     GetUav()->GetVrpnObject()->GetPosition(vrpn_pos);
263     vrpn_pos.To2Dxy(posHold);
264
265     uX->Reset();
266     uY->Reset();
267     behaviourMode=BehaviourMode_t::PositionHold;
268     SetOrientationMode(OrientationMode_t::Custom);
269     Thread::Info("SquareFollower: holding position\n");
```

Illustration 17 : SquareFollower.cpp (method VrpnPositionHold)

From line 259 to 263, we extract the position and the yaw angle from the VRPN. These values are stored in *yawHold* and *posHold* which are attributes of the SquareFollower class.

```
61          flair::filter::Pid *uX, *uY;
62
63          flair::core::Vector2D posHold;
64          float yawHold;
```

Illustration 18 : SquareFollower.h

The attributes *uX* and *uY* represent control laws on the x and y axis. The control law used here is a PID (proportional–integral–derivative). You can find further information here: https://en.wikipedia.org/wiki/PID_controller
At the end of the method *VrpnPositionHold*, we reset the internal state of the control laws.

5) PositionValues

This method is used to get the position error, the velocity error and the reference yaw angle of the drone from the VRPN. These errors will be used to control the drone and stabilize it at the desired setpoint with the control laws *uX* and *uY* introduced just before.

```
129     Vector3D uav_pos,uav_vel; // in VRPN coordinate system
130     Vector2D uav_2Dpos,uav_2Dvel; // in VRPN coordinate system
131
132     GetUav()->GetVrpnObject()->GetPosition(uav_pos);
133     GetUav()->GetVrpnObject()->GetSpeed(uav_vel);
134
135     uav_pos.To2Dxy(uav_2Dpos);
136     uav_vel.To2Dxy(uav_2Dvel);
137
138     if (behaviourMode==BehaviourMode_t::PositionHold) {
139         pos_error=uav_2Dpos-posHold;
140         vel_error=uav_2Dvel;
141         yaw_ref=yawHold;
```

Illustration 19 : SquareFollower.cpp (method PositionValues)

First, we extract the current position and velocity from the VRPN (lines 132 to 136). Then, from lines 139 to 141, if the drone is no longer in motion (holding position), we set the position error, the velocity error and the yaw angle to the values obtained from the VRPN subtracted to the values given by the method *VrpnPositionHold* which is called when the drone is in holding position. If there is no error, *pos_error* must be equal to 0.

```
142     } else { //square
143         Vector3D target_pos;
144         Vector2D square_pos,square_vel;
145         Vector2D target_2Dpos;
146 
147         targetVrpn->GetPosition(target_pos);
148         target_pos.To2Dxy(target_2Dpos);
149         square->SetCenter(target_2Dpos);
150 
151         //square reference
152         square->Update(GetTime());
153         square->GetPosition(square_pos);
154         square->GetSpeed(square_vel);
155 
156         //error in optitrack frame
157         pos_error=uav_2Dpos-square_pos;
158         vel_error=uav_2Dvel-square_vel;
159         yaw_ref=atan2(target_pos.y-uav_pos.y,target_pos.x-uav_pos.x);
160     }
```

Illustration 20 : SquareFollower.cpp (method PositionValues)

If the drone is not in holding position, it is moving to make a square.
From line 147 to 149, the position of the target (character represented by a ninja) is extracted and used to set the center of the square because the drone has to move around this target.
In line 152, the square trajectory is updated which means that we compute the new position of the drone at this specific time.
From line 152 to 154, we get the position and the velocity of the drone that we computed just before.
Then, in lines 157 and 158, we compute the position and velocity errors. The errors are zero if there is no difference between the position and velocity of the drone that we computed previously and the values extracted from the Optitrack (VRPN).

In line 159, we compute the reference yaw angle with the positions of the drone and the target. Remember that the yaw angle must be compute so that the drone looks and follows the target.

```
162     //error in uav frame
163     Quaternion currentQuaternion=GetCurrentQuaternion();
164     Euler currentAngles;//in vrpn frame
165     currentQuaternion.ToEuler(currentAngles);
166     pos_error.Rotate(-currentAngles.yaw);
167     vel_error.Rotate(-currentAngles.yaw);
```

Illustration 21 : SquareFollower.cpp (method PositionValues)

From line 163 to 167, we get the current yaw angle from the current quaternion. Remember that unlike pitch and roll angles which are given by IMU's sensors, yaw angle is extracted from the VRPN and then added to the quaternion.
Here, we use the given yaw angle to rotate the position and the velocity of the drone. As we are in uav coordinate system, we must take the additive inverse (-) of the yaw angle because the z-axis goes down.

6) GetReferenceOrientation

The method *GetReferenceOrientation* is used to get the reference orientation of the drone within an AHRS class. This method is inherited from the *UavStateMachine* class.
The difference between the method *GetOrientation* and *GetReferenceOrientation* is that the first one provides the current orientation even if this orientation is not really the orientation expected whereas the second method provides the reference orientation, i.e. the orientation we want the drone to follow.

```
107     Vector2D pos_err, vel_err; // in Uav coordinate system
108     float yaw_ref;
109     Euler refAngles;
110
111     PositionValues(pos_err, vel_err, yaw_ref);
112
113     refAngles.yaw=yaw_ref;
114
115     uX->SetValues(pos_err.x, vel_err.x);
116     uX->Update(GetTime());
117     refAngles.pitch=uX->Output();
118
119     uY->SetValues(pos_err.y, vel_err.y);
120     uY->Update(GetTime());
121     refAngles.roll=-uY->Output();
122
123     customReferenceOrientation->SetQuaternionAndAngularRates(refAngles.ToQuaternion(),Vector3D(0,0,0));
124
125     return customReferenceOrientation;
```

Illustration 22 : SquareFollower.cpp (method GetReferenceOrientation)

In line 111, the previous method *PositionValues* is used to get the position and velocity errors as well as the reference yaw angle.
From line 115 to 117, we use the control law (PID) uX with the position and velocity errors to get the desired reference pitch angle.
From line 115 to 117, we use the control law (PID) uY with the position and velocity errors to get the desired reference roll angle.
More precisely, *pos_err* and *vel_err* respectively represent the values that will be multiplied with the proportional and derivative terms of the PID.

Remember the uav coordinate system : x-axis forward, y-axis right and z-axis down. As the roll is around the x-axis, it generates a movement along the y-axis and as the pitch is around the y-axis, it generates a movement along the x-axis. That is why the PID related to the x-axis (uX) returns the corrected pitch and the PID related to the y-axis (uY) returns the corrected roll.
Moreover, a positive roll involves a positive y-motion and a positive pitch involves a negative x-motion. That is why, at line 121, we need to get the additive inverse (-) of the roll angle.

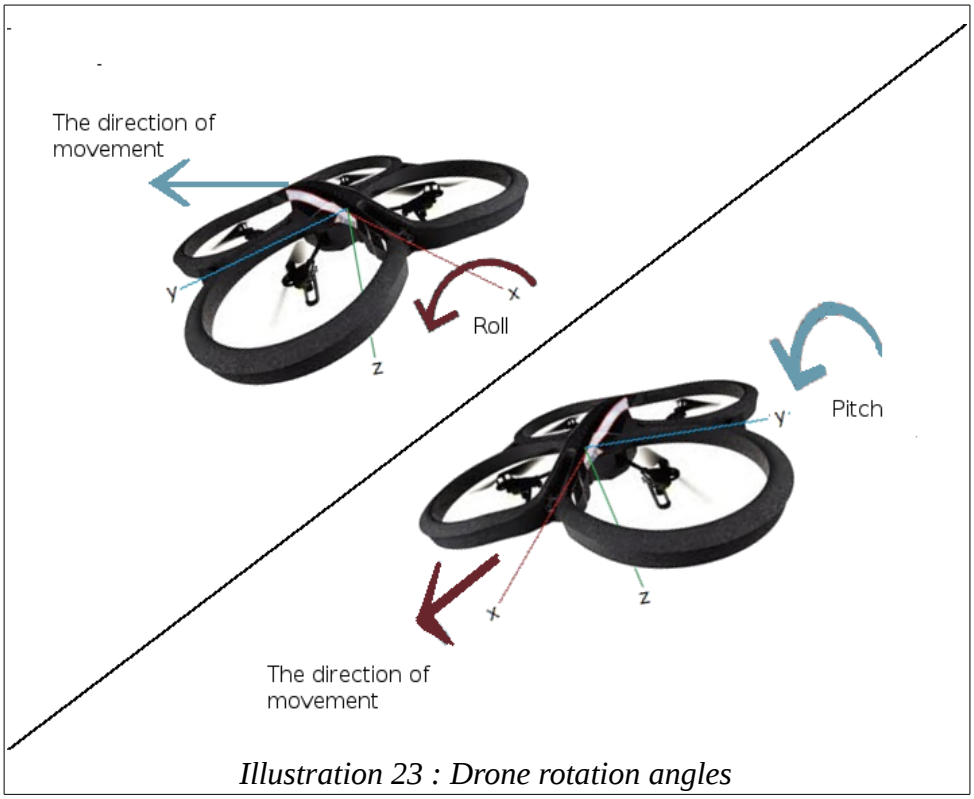


Illustration 23 : Drone rotation angles

7) SignalEvent

The method *SignalEvent* is used to handle events associated with the drone. This method is inherited from the *UavStateMachine* class. Remember that all the possible states of the drone are represented by a state machine.

For example, at the beginning, the drone is in the state *Stopped*. Clicking on the push button *take_off* will transfer it to the state *TakingOff* and when it will reach the desired altitude it will be in the state *Stabilized*. Clicking on the button *start_square* will transfer it to the state *EnteringControlLoop*, etc. At the end, the drone will return to the state *Stopped*.

8) ExtraSecurityCheck

This method is used just for safety in case the VRPN connection with the target or the uav is lost. In that case, the drone enters the state *EnteringFailSafeMode* and automatically lands. This method is inherited from the *UavStateMachine* class.

9) ExtraCheckPushButton

This method is used to start or stop the square if the user respectively clicks on *start_square* or *stop_square.* This method is inherited from the *UavStateMachine* class.

10) ExtraCheckJoystick

Like the previous method, this one is used to start or stop the square with the dualshock3 controller. A chapter about this controller will give you further information. This method is inherited from the *UavStateMachine* class.

11) StopSquare

This method is used to stop the square trajectory.

12) StartSquare

This method will start the square trajectory.

```
227     if (SetOrientationMode(OrientationMode_t::Custom)) {
228         Thread::Info("SquareFollower: start square\n");
229     } else {
230         Thread::Warn("SquareFollower: could not start square\n");
231         return;
232     }
233     Vector3D uav_pos,target_pos;
234     Vector2D uav_2Dpos,target_2Dpos;
235
236     targetVrpn->GetPosition(target_pos);
237     target_pos.To2Dxy(target_2Dpos);
238     square->SetCenter(target_2Dpos);
239
240     GetUav()->GetVrpnObject()->GetPosition(uav_pos);
241     uav_pos.To2Dxy(uav_2Dpos);
242     square->StartTraj(uav_2Dpos);
243
244     uX->Reset();
245     uY->Reset();
246     behaviourMode=BehaviourMode_t::Square;
```

Illustration 24 : SquareFollower.cpp (method StartSquare)

First, note that the lines 228 and 230 are used to display relevant information in the user terminal.
From line 236 to 238, the target position (character, ninja) is extracted and used to set the center of the square.
From line 240 to 242, the drone position is extracted and used to start the square trajectory.
Remember that the attribute *square* is a *TrajectoryGenerator2DSquare* object. This class will be discussed in the next section.
Finally, in line 246, the drone behavior mode is set to *Square* because at this moment the drone is starting its trajectory.

e) TrajectoryGenerator2DSquare implementation

In that section, we are going to see how concretely is implemented the square trajectory.
The source files *TrajectoryGenerator2DSquare.h* and *TrajectoryGenerator2DSquare.cpp* can be found in *$FLAIR_ROOT/flair-src/lib/FlairFilter/src.* They are not very difficult to understand so we will not focus on them. One important thing is that this class uses another class to implement the square equations. This technique is called ***"Pimpl"*** or ***"Pointer to implementation"***. It is used to remove implementation details of a class from its object representation by placing them in a separate class, accessed through an opaque pointer. More details here : http://wiki.c2.com/?PimplIdiom.

```
32  TrajectoryGenerator2DSquare::TrajectoryGenerator2DSquare(
33      const LayoutPosition *position, string name)
34      : IODevice(position->getLayout(), name) {
35    pimpl_ = new TrajectoryGenerator2DSquare_impl(this, position, name);
36    AddDataToLog(pimpl_->output);
37  }
```

Illustration 25 : TrajectoryGenerator2DSquare.cpp (Constructor)

At line 35, a pointer to the square implementation class is created.

f) TrajectoryGenerator2DSquare_impl class

This class generates a square trajectory for the previous class *TrajectoryGenerator2DSquare.*
The file *TrajectoryGenerator2DSquare_impl.cpp* can be found in *$FLAIR_ROOT/flair-src/lib/FlairFilter/src* but its header *TrajectoryGenerator2DSquare_impl.h* is in *$FLAIR_ROOT/flair-src/lib/FlairFilter/src/unexported.* All classes inside the repertory *unexported* does not require recompiling (see *"**Pimpl**" design pattern*).
Now we are going to briefly explain the methods of this class.

1) Constructor

```
34  TrajectoryGenerator2DSquare_impl::TrajectoryGenerator2DSquare_impl(
35      TrajectoryGenerator2DSquare *self, const LayoutPosition *position,
36      string name) {
37    first_update = true;
38    is_running = false;
39
40    nb = 0;
41
42    // init UI
43    GroupBox *reglages_groupbox = new GroupBox(position, name);
44
45    distance = new DoubleSpinBox(reglages_groupbox->LastRowLastCol(), "D", " m", 0,
46                                 1000, .1);
47    velocity = new DoubleSpinBox(reglages_groupbox->LastRowLastCol(), "velocity",
48                                 " m/s", -10, 10, 1);
49    acceleration =
50        new DoubleSpinBox(reglages_groupbox->LastRowLastCol(),
51                          "acceleration (absolute)", " m/s²", 0, 10, .1);
```

Illustration 26 : TrajectoryGenerator2DSquare_impl.cpp (Constructor)

At lines 37, 38, 40 we set the value of some attributes we are going to see later.
From line 43 to 51, the user interface of the ground control station is extended with new double spin boxes. The first one is used to set the length of a side of the square (*distance*), the second represents the setpoint speed and the third is the setpoint acceleration.

```
53      // init matrix
54      cvmatrix_descriptor *desc = new cvmatrix_descriptor(2, 2);
55      desc->SetElementName(0, 0, "pos.x");
56      desc->SetElementName(0, 1, "pos.y");
57      desc->SetElementName(1, 0, "vel.x");
58      desc->SetElementName(1, 1, "vel.y");
59      output = new cvmatrix(self, desc, floatType, name);
60
61      output->SetValue(0, 0, 0);
62      output->SetValue(0, 1, 0);
63      output->SetValue(1, 0, 0);
64      output->SetValue(1, 1, 0);
```

Illustration 27 : TrajectoryGenerator2DSquare_impl.cpp (Constructor)

From lines 54 to 59, a (2x2) matrix is constructed : $\begin{pmatrix} x & y \\ \dot{x} & \dot{y} \end{pmatrix} = \begin{pmatrix} pos.x & pos.y \\ vel.x & vel.y \end{pmatrix}$

Then from lines 61 to 64, the matrix is initialized to 0.

2) StartTraj

```
71  void TrajectoryGenerator2DSquare_impl::StartTraj(const Vector2D &start_pos,
72                                                   float nb_lap) {
73    is_running = true;
74    first_update = true;
75    this->nb_lap = nb_lap;
76    nb = 0;
77
78    // configure trajectory
79    posStart.x = start_pos.x - pos_off.x;
80    posStart.y = start_pos.y - pos_off.y;
81    CurrentTime = 0;
82    FinishTime = 0;
83  }
```

Illustration 28 : TrajectoryGenerator2DSquare_impl.cpp (StartTraj)

The method *StartTraj* is called when the square trajectory is launched.

From lines 73 to 76, some attributes are initialized. The attribute *is_running* is used to know if the trajectory has been launched, *first_update* is used at the beginning of the trajectory to define time as we will see later, and *nb* is used to count the number of translations.
In lines 79 and 80, we get the starting position of the drone. Here *pos_off* represents the center of the square i.e. the position of the target to follow (ninja). This value is defined in the method *SetCenter* of the *TrajectoryGenerator2DSquare* class.

3) Update

This method is used to update the square trajectory at each instant. It is called every time during the trajectory. The goal of this function is to compute the trajectory at each instant *T = [t0, t1, t2, …, tn]* because the trajectory is defined by a set of functions depending on a time variable *t*. At *t0*, the trajectory is calculated then at *t1* it is recalculated until *tn*.
It should be remembered that time is in nanosecond.

```
85  void TrajectoryGenerator2DSquare_impl::Update(Time time) {
86    float delta_t;
87    float V = veloctity->Value();
88    float A = acceleration->Value();
89    float D = distance->Value();
90    Vector2D v;
91
92    if (V < 0)
93      A = -A;
94
95      if (first_update) {
96        first_update = false;
97        previous_time = time;
98
99        return;
100     } else {
101       delta_t = (float)(time - previous_time) / 1000000000.;
102     }
103
104   previous_time = time;
105   CurrentTime += delta_t;
```

Illustration 29 : TrajectoryGenerator2DSquare_impl.cpp (Update)

First, we need to extract the setpoint velocity, the setpoint acceleration and the distance from the ground control station as it is done in lines 87, 88 et 89.

Then, from lines 95 to 101, we check if it is the first update, i.e., the first time the method is called at the very beginning of the trajectory. If so, then we save the current time in *previous_time* and we exit the function. Otherwise, we calculate the difference between current time and previous time in order to get the elapsed time between the previous and the current updates. The difference is converted into second.
Then, in line 105, *CurrentTime* is updated with this difference.

The second part of the method is the implementation of the square trajectory in terms of equations. Here, we will only explain the principle and introduce the equations used.
First of all, five translations are needed to perform the trajectory. After each translation, the attribute *nb,* introduced before, is incremented. The first translation will move the drone to its starting position at the coordinates (D, D). The four last translations will move the drone around the square, first at the coordinates (D, -D), then (-D, -D), then (-D, D) and finally (D,D). The linear motion type here is a uniform linear motion with a constant acceleration defined by the following equations :

$$\begin{cases} v(t) = A * t + v(0) = A * t \\ x(t) = v(t) * t + x(0) \end{cases}$$

To illustrate we are going to see how the second translation is implemented.

```
169             else if(nb == 1)
170             {
171                 v.y = A * (CurrentTime - FinishTime);
172                 if (fabs(v.y) > fabs(V)) {
173                 if (v.y > 0)
174                     v.y = V;
175                   else
176                     v.y = -V;
177                 }
178                 pos.y = -v.y * (CurrentTime - FinishTime) + D;
179                 if (D - v.y * v.y / (2 * A) <= pos.y && v.y >= 0)
180                   A = -A;
181                 if (D - v.y * v.y / (2 * A) >= pos.y && v.y < 0)
182                   A = -A;
183 
184                 if ( (pos.y <= -D && v.y >= 0) || (pos.y >= -D && v.y < 0) )
185                 {
186                   v.y = 0;
187                   pos.y = -D;
188                   nb = 2;
189                   FinishTime = CurrentTime;
190                 }
191 
192             }
```

Illustration 30 : TrajectoryGenerator2DSquare_impl.cpp (Update)

In line 171, the velocity along the y-axis is computed. Remember that this is a new translation with new equations, so current time must be subtracted to the ending time (*FinishTime*) of the previous translation. As time is relative to a single translation we must reset the time at the end of each translations.
From lines 172 to 177, we must check if the absolute value of the current velocity is higher than the setpoint velocity V. If so, the current velocity takes the setpoint value V or -V according to its sign.
Then, in line 178, the position along the y-axis is computed. Here y(0) is equal to D because at the end of the first translation, i.e., at the beginning of the second translation the drone is at the coordinates (D, D) = (x(0), y(0)).
From lines 179 to 182, we slow down the drone if it has traveled half of the path (D/2) by taking the opposite acceleration. We should not suddenly stop the drone.
Finally, from lines 184 to 190, we check if the drone has reached its position (D, -D). If so, the drone is stopped a moment (line 186) and the next translation is launched (line 188).

```
284     // on prend une fois pour toute les mutex et on fait des accès directs
285     output->GetMutex();
286     output->SetValueNoMutex(0, 0, pos.x + pos_off.x);
287     output->SetValueNoMutex(0, 1, pos.y + pos_off.y);
288     output->SetValueNoMutex(1, 0, v.x + vel_off.x);
289     output->SetValueNoMutex(1, 1, v.y + vel_off.y);
290     output->ReleaseMutex();
291
292     output->SetDataTime(time);
293   }
```

Illustration 31 : TrajectoryGenerator2DSquare_impl.cpp (Update)

After having computed the desired position and velocity, we need to insert them into our matrix. This matrix is a critical section, i.e., it can not be access at the same time by several threads. More information about threads here :
https://en.wikipedia.org/wiki/Thread_(computing)

We need to control this access with a mutual exclusion :
https://en.wikipedia.org/wiki/Mutual_exclusion

In order to do so, we can use a lock or mutex :
https://en.wikipedia.org/wiki/Lock_(computer_science)

First, in line 285, the mutex is locked, i.e., the current thread can safely access the matrix. When all data are inserted, the mutex is unlocked.
Here the difference between the functions *SetValue* used in the constructor and *SetValueNoMutex* is that the first one locks and unlocks a mutex for each access whereas the second function does not use a mutex, but it should be called after a mutex locked to avoid conflicts between threads. This function is useful when multiple successive access are done to the elements of the matrix, like in lines 286 to 289, to avoid unnecessary locking and unlocking.

Furthermore, the position of the drone depends on the position of the target (ninja) which is able to move inside the room. Therefore, the current position of the drone (*pos.x/pos.y*) must be added to the current position of the target (*pos_off.x/pos_off.y*). For example, if the drone is performing its square trajectory and if at this specific time it is at the coordinates *pos = (0, 2)* and the target is at the coordinates *pos_off = (0, 0)* , the drone will be at the position *(0, 2) + (0, 0) = (0, 2)*. However if at this specific time the target moves to the coordinates *(2, 0)*, the position of the drone will be *(0, 2) + (2, 0) = (2, 2)*. The target will always be followed ! This is the same for the velocity.

g) Program execution

Now that everything is set up we can launch our program and have a small introduction to the ground control station.

First, launch the ground control station in one terminal :

```
$ $FLAIR_ROOT/flair-bin/tools/scripts/launch_flairgcs.sh
```

In a second terminal, launch the simulator :

```
$ cd $FLAIR_ROOT/flair-src/demos/SquareFollower/simulator/build/bin
$ ./simulator_x4.sh
```

Finally, in a third terminal, launch the uav program :

```
$ cd $FLAIR_ROOT/flair-src/demos/SquareFollower/uav/build/bin
$ ./x4.sh
```

Now, in the ground control station you can see two tabs, one for the simulator and the second for the drone (*x4_0*). The first tab is used to change some settings related to the drone or the target such as their initial positions or their physical and dynamical characteristics.

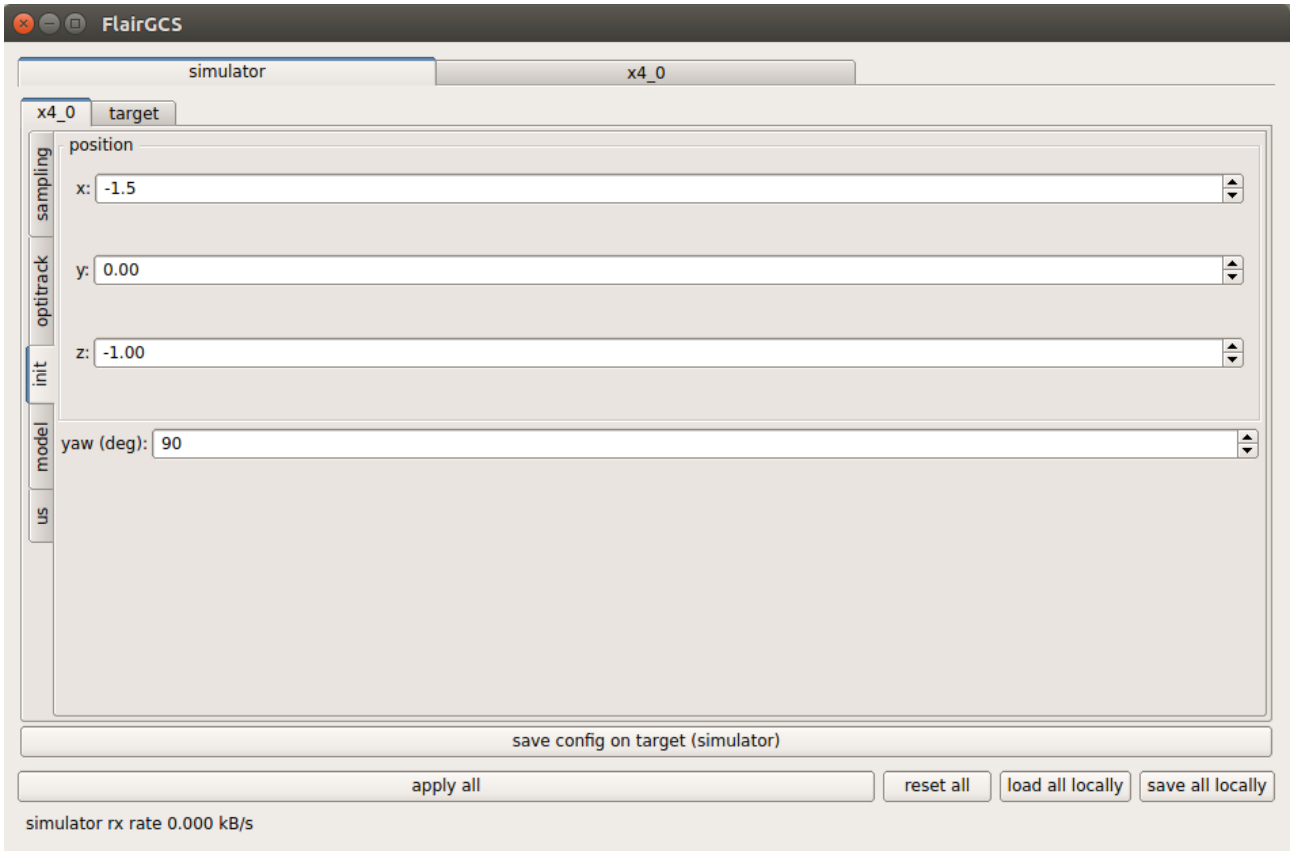


Illustration 32 : Changing drone's initial position

In the second tab (*x4_0*), a sub-tab called *vrpn* is used to set up parameters related to your application such as the length of the side of the square (D) to perform with a given velocity and acceleration. You must be aware that changing these parameters during program execution will have an effect on the drone.

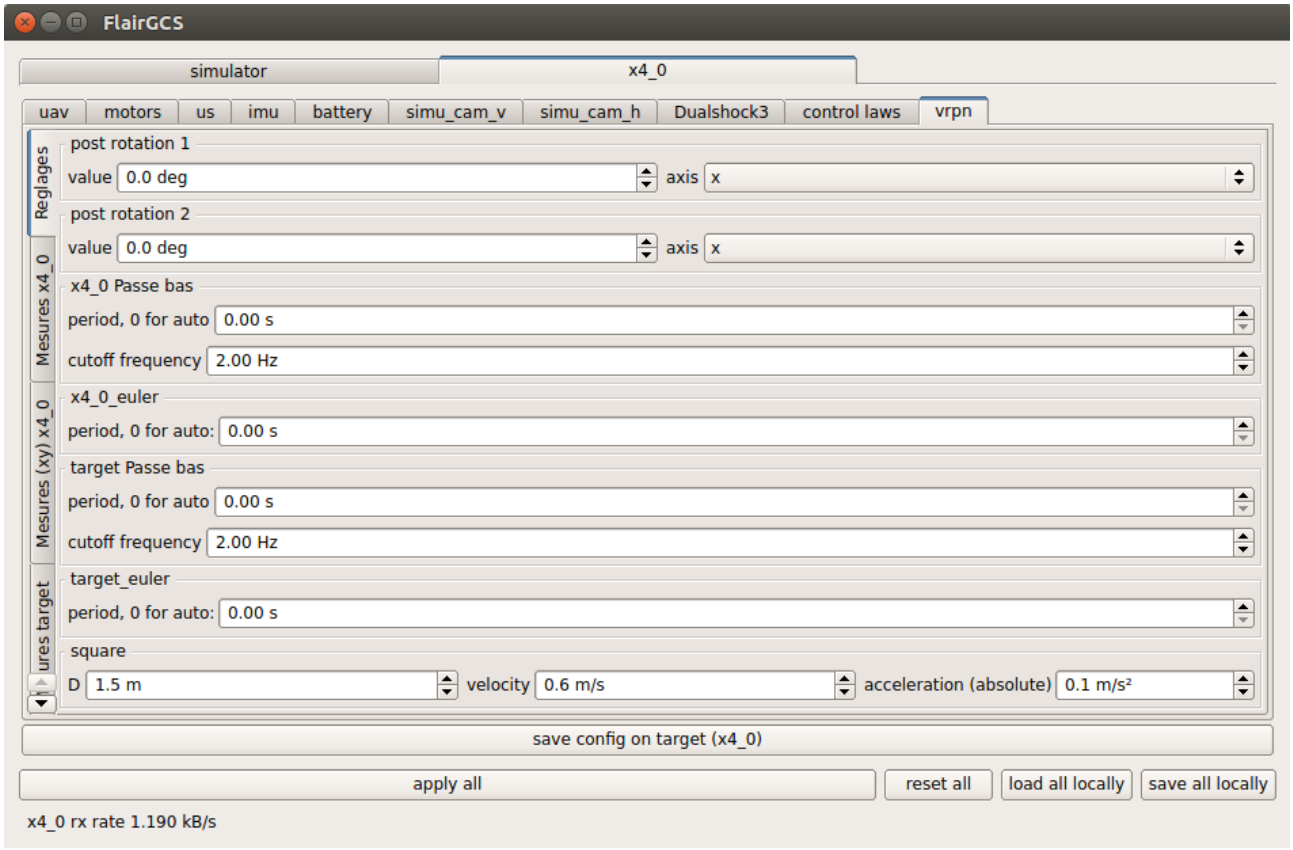


Illustration 33 : Setting up program's parameters

The sub-tab called *uav* is used to start the program. First click on *take_off* then when the drone is stable click on *start_square* to start the trajectory and *stop_square* to stop it.

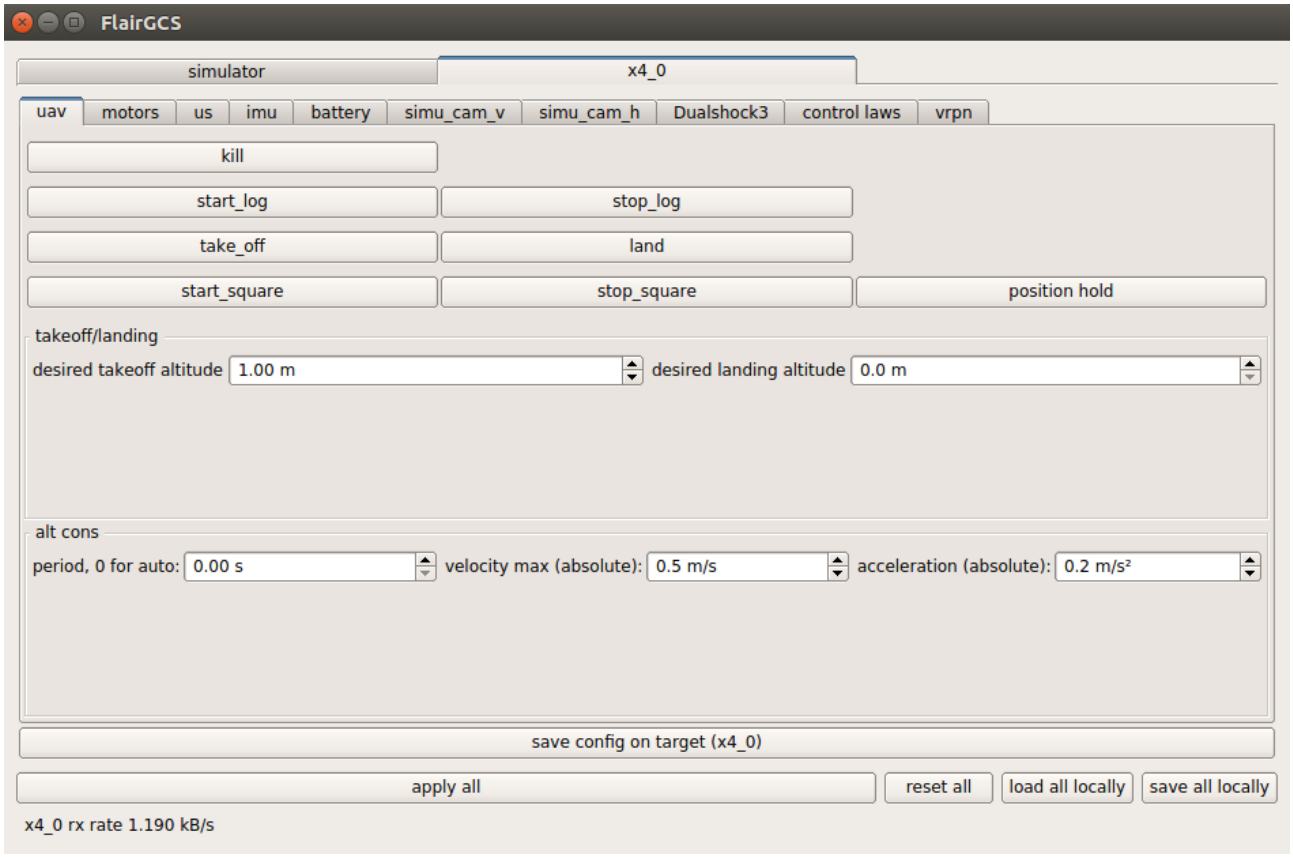


Illustration 34 : Start and stop the square

You can also visualize the position and velocity curves of the drone in the sub-tab *vprn/ Mesures x4_0*.

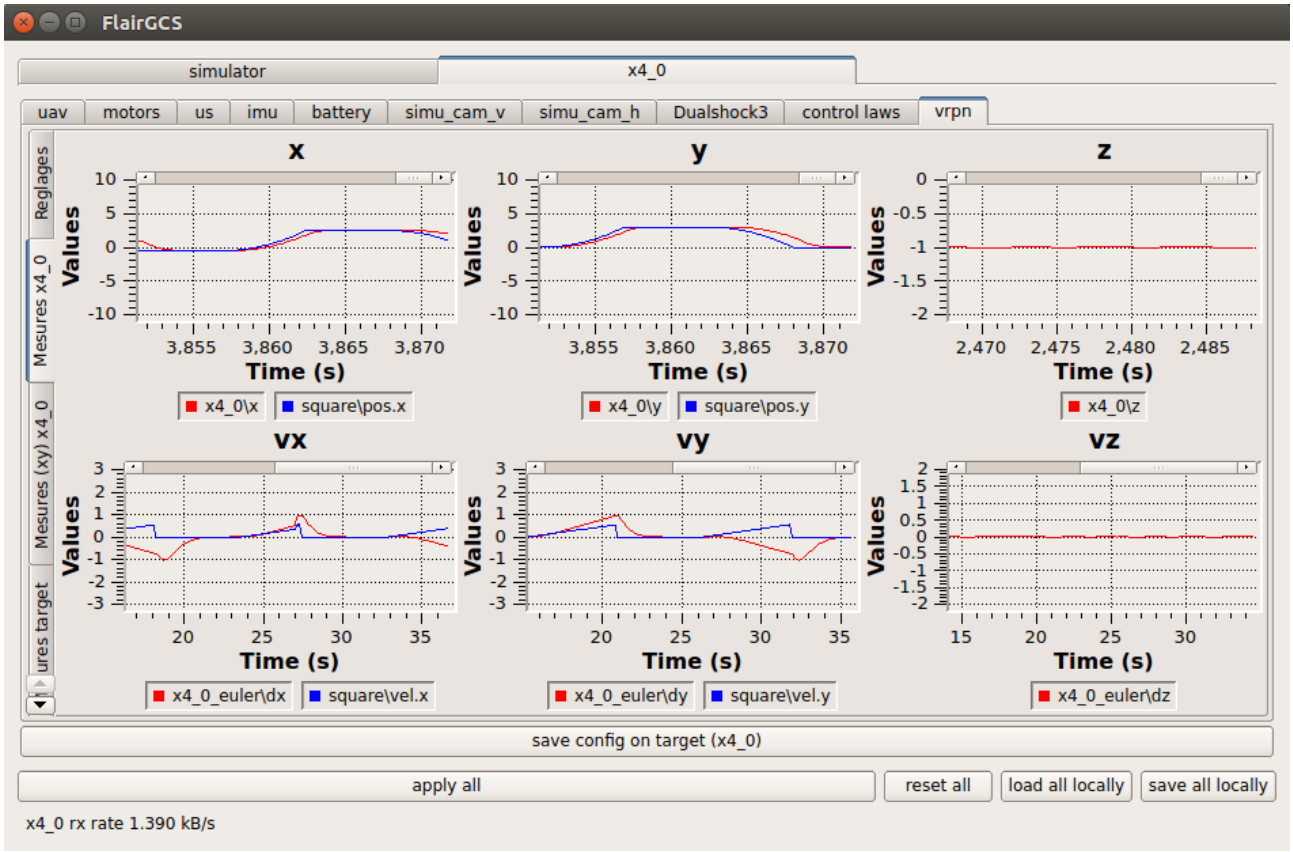


Illustration 35 : Position and velocity curves

In the sub-tab called *imu* you have all the graphics related to the IMU such as roll, yaw and pitch angles or the value of the quaternion.

Illustration 36 : IMU graphics

You could also test different control laws with specific gain in the sub-tab called *control laws*.

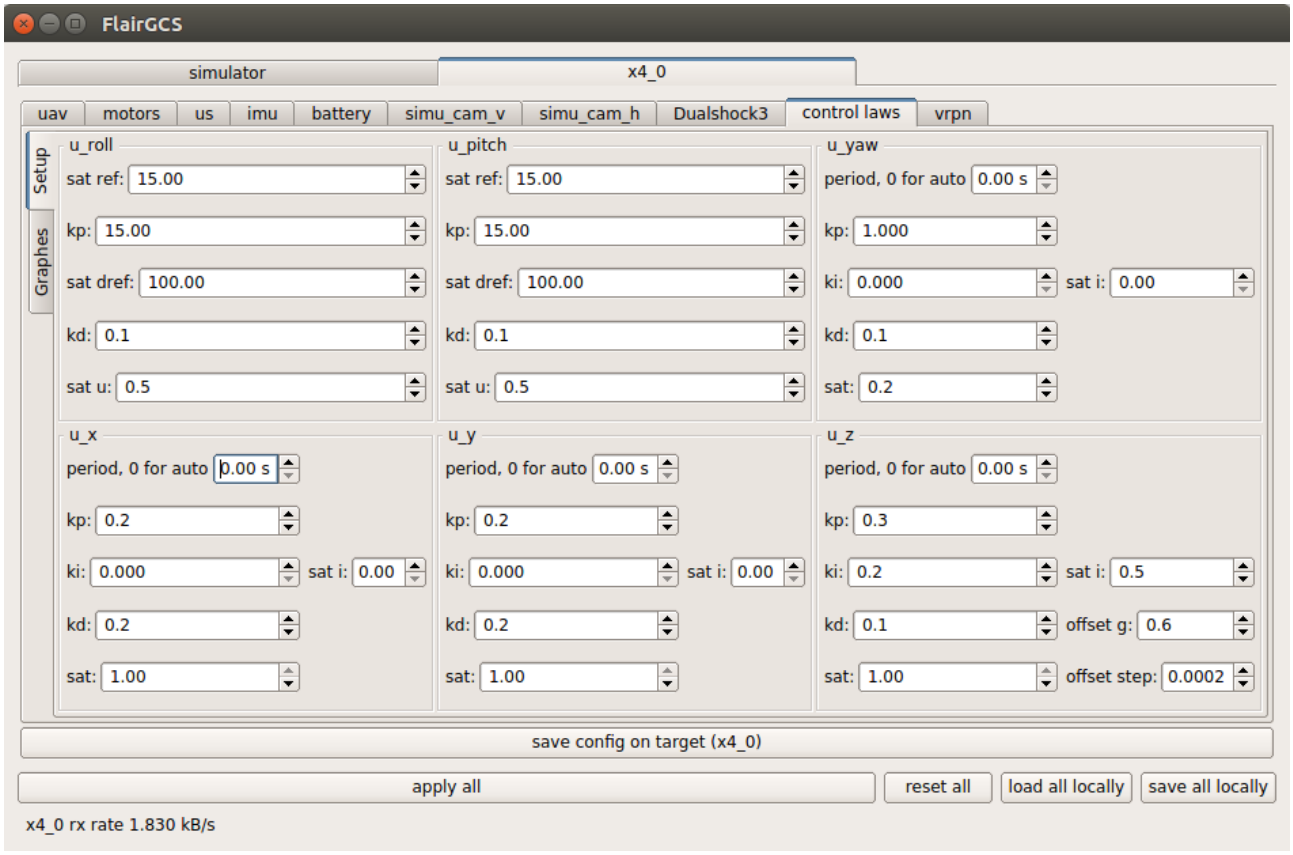


Illustration 37 : Control laws

IV) Using several drones

Now that you are able to perform a square trajectory with a single drone. We are going to see in this part how to use a fleet of two drones. Both drones will perform the square trajectory in parallel. The program can be found in *$FLAIR_ROOT/flair-src/demos/SquareFleet*.

a) Simulator

The *main.cpp* file of the *SquareFleet* simulator is a bit different from that of *SquareFollower* because we need to display two drones.

```
 97    for(int i=0; i<2;i++) {
 98      stringstream uavName;
 99      uavName << name.c_str() << "_" << i;
100      Model *drone=new X8(uavName.str(),i);
101      #ifdef GL
102      SimuUsGL *us_gl=new SimuUsGL(drone,"us",i,0);
103      #endif
104      SimuImu *imu=new SimuImu(drone,"imu",i,0);
105    }
```

Illustration 38 : main.cpp

For each drones, we must create an instance of the classes *Model*, *SimuUsGL* and *SimuImu*.

b) Uav

1) Directory

If you take a look at the launching scripts in *$FLAIR_ROOT/flair/flair src/demos/SquareFleet/uav/build/bin*, you will see that unlike the SquareFollower repertory two scripts are provided instead of one. Indeed, each drone will have its own program i.e. its own launching script. The drones will no be directly synchronized but they will be indirectly through the ground control station.

Clicking on *TakeOff* or *Land* will have an effect on both drones. However, the drones are independent each other and only controlled by the ground control station, therefore they may collide if they are reaching the same point or if a drone is faster than the other while they are doing a translation.
So, both launching scripts will have to be executed in two different terminals (after having launched the ground control station and the simulator).

2) SquareFleet implementation

SquareFleet.cpp is not very different from *SquareFollower.cpp* because as said previously, both drones have their own programs and their own trajectory.

In that example, we want our drones to perform a kind of "choreography" summarized by the following steps:

1. Both drones perform a first square then they stop for a while.
2. Both drones move along the y-axis (→).
3. Then they come back at their starting positions (←) and they stop for a while.
4. Both drones perform a second square.
5. Both drones land.

First of all, to perform both squares we need the coordinates of the squares' centers *(x,y)* assumed to be identical. We also need the distance of the second translation along the y-axis.

```
67      xSquareCenter=new DoubleSpinBox(vrpnclient->GetLayout()->NewRow(),"x square center"," m",-5,5,0.1,1,0);
68      ySquareCenter=new DoubleSpinBox(vrpnclient->GetLayout()->NewRow(),"y square center"," m",-5,5,0.1,1,0);
69      yDisplacement=new DoubleSpinBox(vrpnclient->GetLayout()->NewRow(),"y displacement"," m",0,2,0.1,1,0);
```

Illustration 39 : SquareFleet.cpp (Constructor)

The methods of *SquareFleet.cpp* are very similar to those of *SquareFollower.cpp*. We are just going to focus on *SignalEvent* because it is used to synchronize the drones.

SignalEvent

We have already seen this method previously, it is used to handle events associated with the drone. Here we have exactly 6 possible events :

- EmergencyStop
- TakingOff
- StartLanding
- EnteringControlLoop
- EnteringFailSafeMode
- ZtrajectoryFinished

```
205       case Event_t::EmergencyStop:
206           message->SendMessage("EmergencyStop");
207           break;
208       case Event_t::TakingOff:
209           //behaviourMode=BehaviourMode_t::Default;
210           message->SendMessage("TakeOff");
211           VrpnPositionHold();
212           behaviourMode=BehaviourMode_t::PositionHold1;
213           break;
214       case Event_t::StartLanding:
215           VrpnPositionHold();
216           behaviourMode=BehaviourMode_t::PositionHold4;
217           message->SendMessage("Landing");
218           break;
```

Illustration 40 : SquareFleet.cpp (SignalEvent)

For each events, a message is display to the user.
The event *TakingOff* is triggered when both drones are taking off and the event *StartLanding* is triggered when they are landing at the end of their "choreography".

```
220     case Event_t::EnteringControlLoop:
221         CheckMessages();
222         if ((behaviourMode==BehaviourMode_t::Square1) && (!square->IsRunning())) {
223             VrpnPositionHold();
224             behaviourMode=BehaviourMode_t::PositionHold2;
225 
226             if(posHold.y>0) {
227                 posHold.y-=yDisplacement->Value();
228             } else {
229                 posHold.y+=yDisplacement->Value();
230             }
231             posWait=GetTime();
232             Printf("PositionHold2 -> PositionHold3\n");
233         }
234         if (behaviourMode==BehaviourMode_t::PositionHold2 && GetTime()>(posWait+3*(Time)1000000000)) {
235             behaviourMode=BehaviourMode_t::PositionHold3;
236             if(posHold.y<0) {
237                 posHold.y+=yDisplacement->Value();
238             } else {
239                 posHold.y-=yDisplacement->Value();
240             }
241             posWait=GetTime();
242             Printf("PositionHold2 -> PositionHold3\n");
243         }
```

Illustration 41 : SquareFleet.cpp (SignalEvent)

```
244         if (behaviourMode==BehaviourMode_t::PositionHold3 && GetTime()>(posWait+3*(Time)1000000000)) {
245             behaviourMode=BehaviourMode_t::Square2;
246             StartSquare();
247             Printf("PositionHold3 -> Square2\n");
248         }
249         if ((behaviourMode==BehaviourMode_t::Square2) && (!square->IsRunning())) {
250             Printf("Square2 -> Land\n");
251             behaviourMode=BehaviourMode_t::PositionHold4;
252             Land();
253         }
254         break;
```

Illustration 42 : SquareFleet.cpp (SignalEvent)

The event *EnteringControlLoop* is important because it used to perform the "choreography". In this function we have four "*if*" which represent the four steps of the "choreography".
The first condition is met when both drones have performed their first square (2^{nd}step). This is verified via the attribute *behaviourMode*. If *behaviourMode* equals *Square1* and the drones are not running (line 222), this means that the first square trajectory is finished. Therefore, the second step is launched and a displacement along the y-axis is executed (lines 226 to 230). In line 232, the current time is saved because we need to stop for a while before the next step.
The second condition is met when both drones have performed their first translation (3^{nd}step). In line 234, we check if 3 seconds have elapsed (3*10^9 nanoseconds) from the

last translation. If so, we perform the second translation and both drones come back at their starting positions (before the first translation, 2nd step).
The third condition (4th step) is met 3 seconds after the second translation (3nd step). A second square is performed by the drones.
Finally, the fourth condition (5th step) is met when the second square is finished. Both drones have finished their "choreography" and land.

```
257       case Event_t::EnteringFailSafeMode:
258           behaviourMode=BehaviourMode_t::Default;
259           break;
260       case Event_t::ZTrajectoryFinished:
261           Printf("PositionHold1 -> Square1\n");
262           StartSquare();
263           behaviourMode=BehaviourMode_t::Square1;
264           break;
```

Illustration 43 : SquareFleet.cpp (SignalEvent)

The event *ZtrajectoryFinished* is used to start the first square (1st step).

V) Waypoint program

In this chapter we will study a program which calculates a trajectory for the drone so that it passes through points (whose coordinates are given by the user) and in an orderly manner. This program is the Waypoint program and its structure is close to the other programs studied until then.
The first section deals mainly on how to create input fields on the GUI, the second part goes deeper, focuses on the analysis of trajectory equations.

```
TrajectoryGenerator2DCircle_impl::TrajectoryGenerator2DCircle_impl(
    TrajectoryGenerator2DCircle *self, const LayoutPosition *position,
    string name) {
  first_update = true;
  is_running = false;
  is_finishing = false;

  // init UI
  GroupBox *reglages_groupbox = new GroupBox(position, name);
  T = new DoubleSpinBox(reglages_groupbox->NewRow(), "period, 0 for auto", " s",
                        0, 1, 0.01);
  rayon = new DoubleSpinBox(reglages_groupbox->LastRowLastCol(), "R", " m", 0,
                            1000, .1);
  veloctity = new DoubleSpinBox(reglages_groupbox->LastRowLastCol(), "velocity",
                                " m/s", -10, 10, 1);
acceleration =new DoubleSpinBox(reglages_groupbox->LastRowLastCol(),
                        "acceleration (absolute)", " m/s²", -10, 10, .1);
     xPoint1=new DoubleSpinBox(reglages_groupbox->NewRow(),"xPoint1","", -100, 100, 1);
    yPoint1=new DoubleSpinBox(reglages_groupbox->LastRowLastCol(),"yPoint1","", -100, 100, 1);
    xPoint2=new DoubleSpinBox(reglages_groupbox->NewRow(),"xPoint2","", -100, 100, 1);
    yPoint2=new DoubleSpinBox(reglages_groupbox->LastRowLastCol(),"yPoint2","", -100, 100, 1);
    xPoint3=new DoubleSpinBox(reglages_groupbox->NewRow(),"xPoint3","", -100, 100, 1);
    yPoint3=new DoubleSpinBox(reglages_groupbox->LastRowLastCol(),"yPoint3","", -100, 100, 1);
    xPoint4=new DoubleSpinBox(reglages_groupbox->NewRow(),"xPoint4","", -100, 100, 1);
    yPoint4=new DoubleSpinBox(reglages_groupbox->LastRowLastCol(),"yPoint4","", -100, 100, 1);
  // init matrix
  cvmatrix_descriptor *desc = new cvmatrix_descriptor(2, 2);
  desc->SetElementName(0, 0, "pos.x");
  desc->SetElementName(0, 1, "pos.y");
  desc->SetElementName(1, 0, "vel.x");
  desc->SetElementName(1, 1, "vel.y");
  output = new cvmatrix(self, desc, floatType, name);
  delete desc;
}
```

Illustration 44 : Constructor

a) Graphical interface

In this part we will see how to create spin-boxes and display them in the graphical interface of the program.
First of all spin-boxes are declared in the header file like this:
*Flair :: gui :: DoubleSpinBox * xPointn, * yPointn;*

Then the spin boxes are initialized in the constructor of *trajectoryGenerator2DCircle_impl* and they are placed in the *reglagles_goupbox* which groups the values reported by the user such as speed or acceleration.

xPoint1=new DoubleSpinBox(reglages_groupbox->NewRow(),"xPoint1","", -100, 100, 1);

The class doublespinbox takes as parameters:

- position : here the *reglages_groupbox* variable allows to create a new row in the form
- A name "*xPoint1*"
- minimal value
- maximal value
- the step

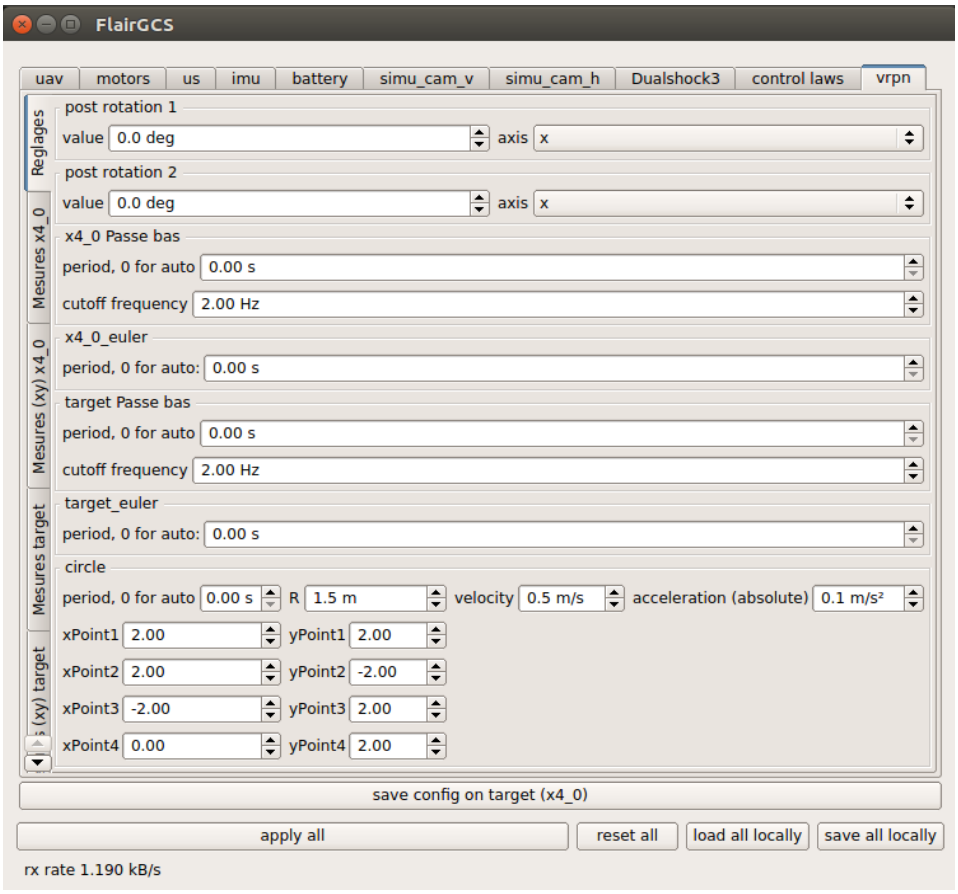


Illustration 45 : Reglage Box with double spin boxes (xPoint , yPoint)

b) Equations of the program

Function to compute the trajectory:

In this program we have created a function that allows to calculate the trajectory by taking as parameters

- The starting coordinates
- The arrival coordinates
- The number of the stage

Each time the drone arrives at a point (xn,yn) on the itinerary the stage number is updated and the finish time becomes equal to the current time and then a new trajectory is calculated between the new starting point (xn,yn) and a new point of arrival (xn+1,yn+1).

The function that calculates the trajectory is declared inside the function *TrajectoryGenerator2DCircle_impl :: Update (Time time)* which is called at each delta t.

```
112  auto calcu = [&] (float xo,float xn, float yo, float yn,int n){
113             D = sqrt((xo-xn)*(xo-xn)+(yo-yn)*(yo-yn));
114                       if (fabs(v.y) > fabs(V)) {
115                           if (v.y > 0)
116                           v.y = V;
117                       else
118                           v.y = -V;
119               }
120                 if (fabs(v.x) > fabs(V)) {
121                       if (v.x > 0)
122                         v.x = V;
123                       else
124                         v.x = -V;
125             }
126
127             pos.x= v.x * (CurrentTime - FinishTime) + xo;
128             pos.y = v.y * (CurrentTime - FinishTime) + yo;
129
```

Illustration 46 : Function calcu

D is the variable representing the distance between the starting point and the point of arrival.
xo and yo are the coordinates of the starting point while xn and yn are the coordinates of the point of arrival. n is the number of the step.

The x-position of the drone during displacement follows the equation x(t) = v.x * t + xo
t = CurrentTime - FinishTime

Finishtime is the ending time of the last step.
The conditions on v.x and v.y are used to keep the speed v.x and v.y lower than V value that was specified in the box_setup seen previously.

```
128
129                 if (yn-v.y * v.y/ (2 * A) <= pos.y && v.y >= 0)
130                   A = -A;
131                 if (yn-v.y * v.y / (2 * A) >= pos.y && v.y < 0)
132                   A = -A;
133                 if (xn-v.x * v.x / (2 * A) <= pos.x && v.x >= 0)
134                   A = -A;
135                 if (xn-v.x * v.x / (2 * A) >= pos.x && v.x < 0)
136                   A = -A;
```

Illustration 47 : Conditions met when reaching half of the distance

V(t)=dx/dt as we have seen x(t)=V*t+xo
A(t)=dV/dt = d^2x/dt^2 so v.x = A * (CurrentTime – FinishTime) ; same for v.y
and x(t)=1/2 $A*t^2$ = ½ (A*t)(A*t)/A=$(v.x)^2/2A$

Here the condition xn-x(t) < pos.x (where pos.x is the real position of the drone on x) means that the drone has passed the middle of the trajectory on x. And v.x >= 0 means that the drone advances so we decelerate it by replacing A by -A. The same thing for the acceleration if one has not reached half and the velocity is less than 0 so we reverse the sign of A so that the drone accelerate.

```
138                     if ( (pos.y >= yn && v.y >= 0) || (pos.y <= yn && v.y < 0) )
139                     {
140                       v.y = 0;
141
142                         pos.y = yn;
143
144                   }
145                if ( (pos.x >= xn && v.x >= 0) || (pos.x <= xn && v.x < 0) )
146                     {
147                       v.x = 0;
148
149                         pos.x = xn;
150
151                   }
152
153                  if ( ((pos.y >= yn && v.y >= 0) || (pos.y <= yn && v.y < 0)) && ((pos.x >= xn && v.x >= 0) || (pos.x <= xn && v.x < 0)) )     {
154               v.x = 0;
155               pos.x = xn;
156             v.y = 0;
157               pos.y = yn;
158               FinishTime = CurrentTime;
159               setNb(n+1);
160
161                }
162
163
```

Illustration 48 : Stopping conditions

The first condition makes it possible to stop the drone at the correct position on the y-axis and the second on the x-axis. The third turns off the drone if it reaches xn and yn and sets the F*inishtime* to C*urrentTime* and increases the step number by 1 with the *setNb* function.

VI) Skeleton and DualShock3 controller

1) How are composed the Skeletons programs?

In this chapter you will familiarize with *skeleton* programs and you will understand how to control a drone with a PS3 controller.
Skeletons are examples included in the flair sources, this programs have to be completed to fit your needs but here we will only talk about parts which are already implemented.
First of all you have to go to the demos folder in the *flair-src* repository. Then in *skeletons,* here you can choose between *customtorque* and *customangle.*
customtorque shows how to set custom torques, to use our own control laws.
And *customangle* shows how to set custom reference angles, and use the default control laws (PID).
In this part we will not talk about control laws as a result this difference doesn't have any importance for the following, you can choose whatever you want.

The skeleton repository is organized as any other FL-AIR program repository, if you want explanations about this organization and the compilation way we invite you to return to the previous chapter "Your first program" because in this section we will only focus on the source code.
In the src folder there are 3 files : *main.cpp* , *MyApp.cpp* and *MyApp.h.*
main.cpp contains the main function that is called at program startup.

The main function is preceded by *parseOption* method that is used to initialize variables declared above.
The main function creates four objects :

- *manager* : a *FrameworkManager* which is the main class of the Framework library, it sets attributes such as the address and the port to setup the connection with the ground control station (line 41 to 45) .

- *drone* : a *UAV* object, its constructor takes as parameters the *FrameworkManger* object created previously, a name and a type which are variables initiazed with the *parseOption* function.

```
using namespace std;
using namespace flair::core;
using namespace flair::meta;
using namespace flair::sensor;

string uav_type;
string log_path;
int port;
int ds3port;
string xml_file;
string name;
string address;

void parseOptions(int argc, char** argv);


int main(int argc, char* argv[]) {
    parseOptions(argc,argv);

    FrameworkManager *manager;
    manager= new FrameworkManager(name);
    manager->SetupConnection(address,port);
    manager->SetupUserInterface(xml_file);
    manager->SetupLogger(log_path);

    Uav* drone=CreateUav(manager,name,uav_type);
    TargetEthController *controller=new TargetEthController(manager,"Dualshock3",ds3port);
    MyApp* app=new MyApp(drone,controller);

    app->Start();
    app->Join();

    delete manager;
}
```

Illustration 49 : Skeleton (main.cpp)

- The third object is a TargetEthController. As the *UAV* object it takes a manager, a name and connection port as parameters. This object is used to connect the Dualshock3 controller , this class inherits *TargetController* a base class for target side remote controls. It includes the connection and the acquisition of data coming from *IODevices* as the functions GetButtonNumber() or IsButtonPressed (unsigned int buttonId) .We will pay attention at this class later in this section.

This 3 classes are included in the Flair Libraries and are detailed in the Flair Documentation you can take a look by following this link :
https://devel.hds.utc.fr/svn/flairdev/tags/latest/doc/Flair/index.html

- The last created object is *MyApp*. This class is used only in the skeleton programs and is almost empty : it contains only the basic methods seen previously in this guide such as the SignalEvent(Event_t event) method that keeps the UAV in default mode for taking off and being able to use safe-mode (for more information look at sections 7, 8 and 9 of the chapter *SquareFollower* implementation).

2) How is the Dualshock3 controller recognized by your computer ?

The framework FLAIR contain a program allowing to connect a PS3 controller to a computer and to interpret the signals from the joystick. The source code of this program is contain in the tool folder in *flair-src* directory. The class *DualShock3* inherited from *HostEthController* the base class for host side remote controls that talks to target side through Ethernet connection.

Here the constructor of *DualShock3 :*

```
49  */
50  class DualShock3 : public HostEthController {
51  public:
52    typedef enum { Usb, Bluetooth } ConnectionType_t;
53
54    DualShock3(const core::FrameworkManager *parent, std::string name,
55               std::string receiverAddress, int receiverPort,
56               ConnectionType_t connectionType, uint32_t period = 10,
57               uint32_t bitsPerAxis = 7, uint8_t priority = 0);
58    ~DualShock3();
59
```

Illustration 50 : Dualshock3 tool (constructor)

Similarly to many Fl-AIR classes, this class takes a *FrameworkManager* object and a *name* as parameters. The *receiverAdress* is the data receiver address (ex: IP address of the UAV), the *reciverPort* is a local port used to connect the controller to the ground station, the *connectionType* is connection type (USB or Bluetooth), *period* is sending data period (frequency of measurement acquisition from the controller).

Now let's see the methods and the functions that compose this class:
First of all we notify that we have *gui* variables that's mean that the program has a graphical interface (ground station).
Some functions are explicit: *GetAxisDescription* and *GetButtonDescription* are functions used to get the axis and buttons names like "left stick x-axis" for axis or "circle" for buttons .
The other *get* functions : *getAxisData* returns the axis values and *getButtonData* returns button values.

```
60  private:
61    gui::SpinBox *deadZone;
62    gui::CheckBox *enabled;
63    gui::Label *batteryChargeLevel;
64    ConnectionType_t connectionType;
65    core::Time now;
66
67    std::string GetAxisDescription(unsigned int axis);
68    std::string GetButtonDescription(unsigned int button);_
69    void GetAxisData();
70    void GetButtonData();
71    bool IsDataFrameReady();
72    void ProcessMessage(core::Message *controllerAction);
73
74    void UpdateFrom(const core::io_data *data){};
75    void fatal(const char *msg);
76    int l2cap_listen(const bdaddr_t *bdaddr, unsigned short psm);
77    struct motion_dev *accept_device(int csk, int isk);
78    void hidp_trans(int csk, char *buf, int len);
79    void setup_device(struct motion_dev *dev);
80    bool parse_report_sixaxis_ds3(unsigned char *r, int len);
81    int mystr2ba(const char *s, bdaddr_t *ba);
82    char *myba2str(const bdaddr_t *ba);
83    int8_t compute_dead_zone(int axis, unsigned char value);
84    struct motion_dev *dev;
85    int usb_fd;
86    int isk;
87    core::Time last_voltage_time;
88
89    int8_t *datas;
90    uint8_t dataSize;
91
92    void usb_scan();
93    void usb_pair_device(struct usb_device *dev, int itfnum);
94
95    void rumble(uint8_t left_force, uint8_t left_timeout, uint8_t right_force,
96                uint8_t right_timeout);
97    void set_led(uint8_t led, uint8_t on_timeout, uint8_t off_timeout);
98    char ledmask;
99    uint8_t led1_on, led1_off, led2_on, led2_off, led3_on, led3_off, led4_on,
100       led4_off;
101 };
102 }
103 }
104
105 #endif // DUALSHOCK3_H
106
```

Illustration 51 : Dualshock3 script class methods

compute_dead_zone(axis, value) : the dead zone is a limitation on the axis values, it is not safe to let the value without any control, this function returns the corrected value and the dead zone is entered in a *SpinBox* (declared at line 61).

3) Using the skeleton program to control the drone with a Dualshock3

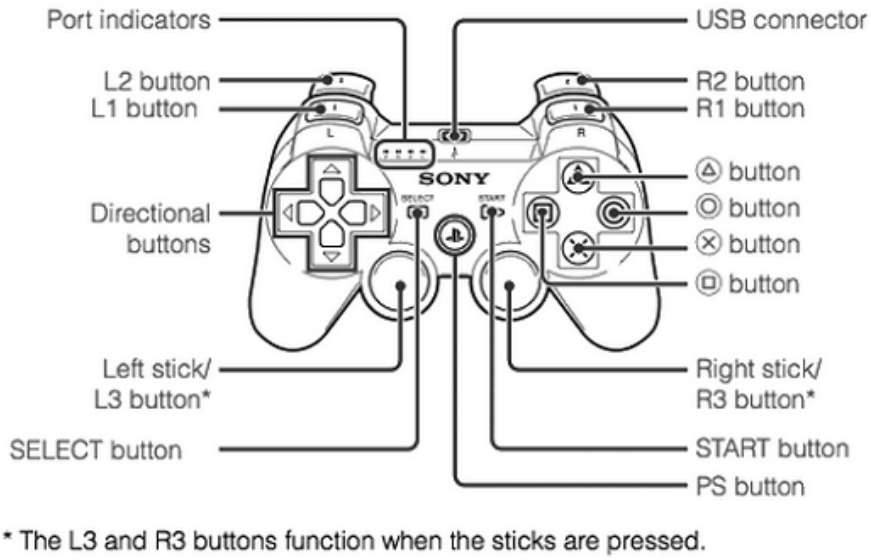


Illustration 52 : Dualshock3 controller

Now that we have seen the skeleton source program and the Dualshock3 tool let's see how to execute these programs and how it works in the simulator.

First, run the ground control station in one terminal :

```
$  $FLAIR_ROOT/flair-bin/tools/scripts/launch_flairgcs.sh
```

Then, plug in your PS3 Joystick to your computer with USB or Bluetooth. Launch the Flair script Dualshock3 in another terminal:

For Bluetooth connection

```
$ $FLAIR_ROOT/flair-bin/tools/scripts/dualshock3_local_bt.sh
```

Or if you prefer USB

```
$ $FLAIR_ROOT/flair-bin/tools/scripts/dualshock3_local_usb.sh
```

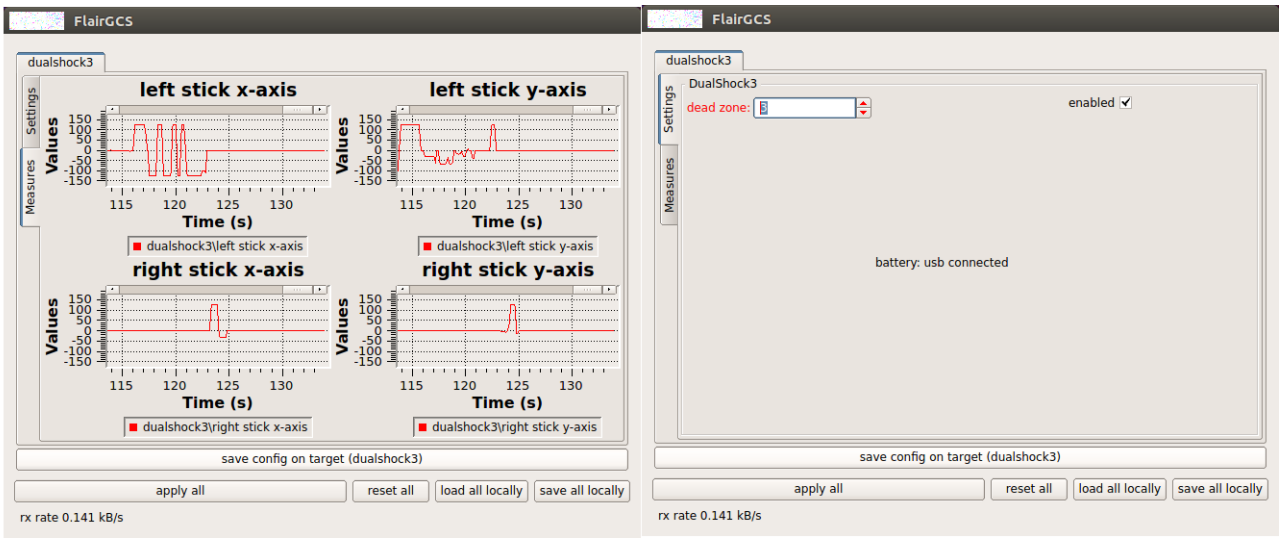
After this you can press the PS button on the controller to turn the controller on as it is shown in this screen-shot:

```
Using FlairCore library:
  -built by sofiane@sofiane-HP on samedi 22 avril 2017, 14:28:09 (UTC+0200)
  -with GCC 4.9.1 from /opt/robomap3/1.7.3/core2-64/sysroots/x86_64-pokysdk-linu
x/usr/bin/x86_64-poky-linux/x86_64-poky-linux-gcc
  -svnversion of /home/sofiane/flair/flair-src/lib/FlairCore is 148

Connecting to 127.0.0.1:9000
System is little endian
successfully opened /dev/hidraw0
Press PS button to turn the controller on
```

Illustration 53 : Controller connection

If you return to the *flairgcs* program you can see a new tab: “DualShock3”
There is a “settings” tab and a “measures” tab : as we saw previously the user can change the dead zone value.
These graphics show the values taken by the different axis when you interact with the joysticks.



Measures of the joysticks axis. *Dead zone setting*

Illustration 54

Now let launch the skeleton UAV and simulator programs:

Open another terminal and execute this command for the simulator:

```
$ cd $FLAIR_ROOT/flair-src/demos/SquareFollower/simulator/build/bin
$  ./simulator_x4.sh
```

and open a last terminal for the uav program:

```
$ cd $FLAIR_ROOT/flair-src/demos/Skeletons/uav/build/bin
$  ./x4.sh
```

We obtain this windows, where we can set the joystick's consigns that have effects on the calibration of each joystick movement. As seen previously the roll is the angle made by the UAV around the X axis so changing "debattemnt roll" parameter will change the impact of the Joystick movement on the roll.

Experimental process changing dead zone values

Illustration 55 : Dead zone value -100

Illustration 56 : Dead zone value 20

We did the same movement with the joysticks but with different dead zone values. The first picture correspond to the lowest dead zone value, we can see that values taken by axis are limited in a smaller area than in the second graph. To conclude a small dead zone reduces the interval that the value of the axis can take.

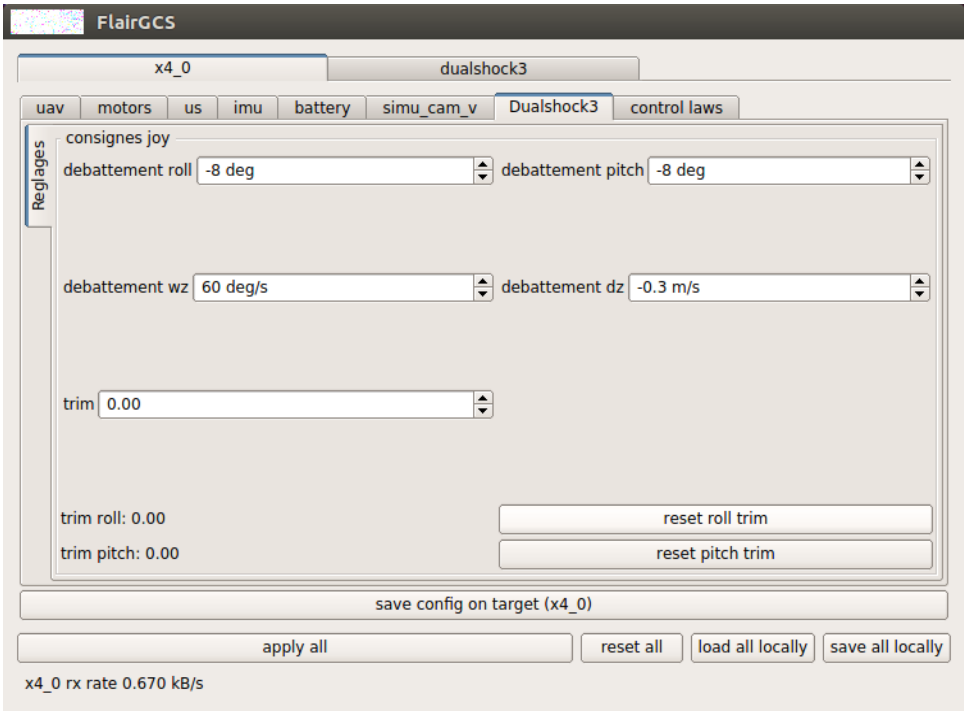


Illustration 57 : X4 program / DualShock3 settings

To start controlling the drone with the controller press start button to take off. Then pilot it with the two joysticks slowly especially if you have a high dead zone value.

To pilot UAV you have two joysticks the left and the right one. Left joystick is for changing the UAV angle on z (yaw) and Right Joystick is for changing the pitch and the roll. Push the cross button if you have to do an emergency stop.

VI) Aviary safety

To manipulate drones it is important to know some safety rules in order to prevent any accident and damage to equipment.

a) How is organized the aviary?

The aviary is composed of two rooms, a control area and a flight zone which are separated by a glass.
The flight zone is surrounded by sensors used to follow drones displacements (Optitrack).
The system is linked to the control station, a dedicated computer is reserved for the software usage.
In the control room there are computers forming the ground control station connected to a local network.
The drones are connected to the ground station by SSH and each drone has a unique IP helping to differentiate one from another. The IP address is written under drones.

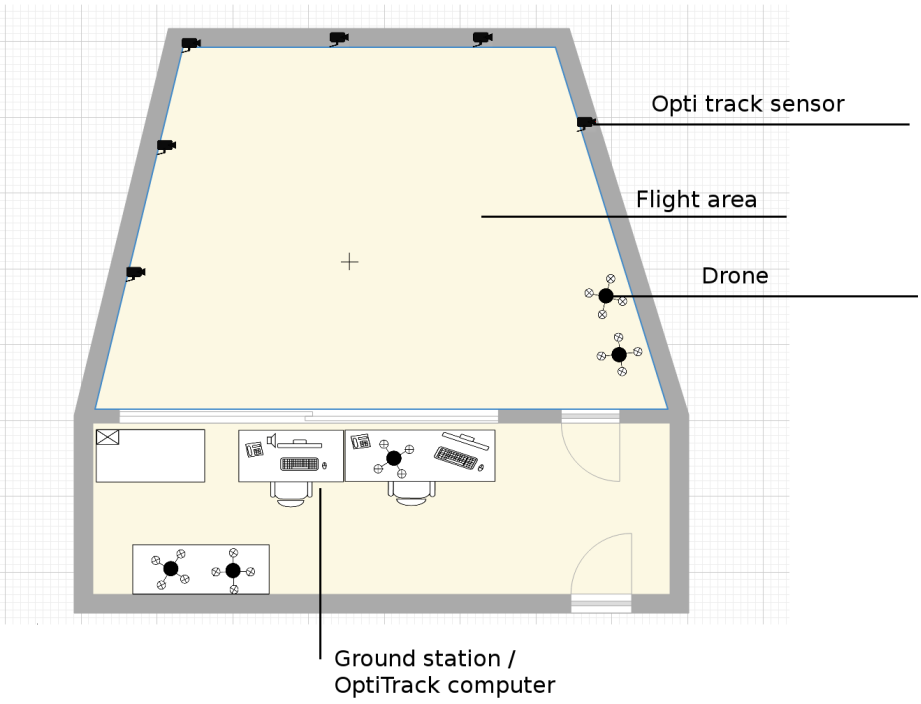


Illustration 58 : Aviary plan

Rules before a flight:

Drones are fragile and very expensive as a result we have to manipulate them carefully. Before each flight you have to be certain that your program is correct and doesn't represent a danger. You must test your program on the flair simulator several times. And check if the program allows to make an emergency stop to avoid any damage for you and for the equipment.
Before starting any flight you have to test the drone's condition by testing each engine with Flair ground control station.
When drones are flying never leave the control area to join the flight zone and if it's unavoidable use your protections: glasses and gloves.
Screenshot motor

Equipment :

To start a flight we have to outfit drones with polystyrene shell to protect the devices. This protective cases are covered by sensors that make the drone visible by the Optitrack device.
The drones are powered by lithium polymer batteries, it's important to check the battery power before using it, you can do that with a multimeter. A battery is considered to be charging or operational if it has a voltage of 12V instead it should not be used if its voltage is below 10V. Batteries can be recharged by a mains chargers.

Since the batteries are lithium batteries it is necessary to take this safety measures :

- not leaving a battery in charge unattended
- not to charge a battery that has just been used
- never leave a battery in a drone.
- store batteries in dedicated bags by separating the loaded and the unloaded ones.

b) Steps to launch a program on a drone

In this chapter we will learn how to launch a program on a drone, we illustrate the process with *CircleFollower* program included on flair demo. Follow this step carefully in the order they are presented:

Connect your drone to your computer

First of all you have to connect your computer to the same WIFI network as your drone and know the IP address of the drone.

The connection to the drone is made through an SSH connection; here is the command line to establish this connection:

```
$ $ ssh root@drone-ip
```

Drone ip address is written on the drone

You are now connected to the drone and you can transfer the program.

Make sure that the program is compiled for an ARM target with this command.

```
$ $ cd $FLAIR_ROOT/flair-src/demos/CircleFollower/uav
  $ $FLAIR_ROOT/flair-dev/scripts/cmake_codeblocks.sh
  $ cd ../build_arm
  $ make
```

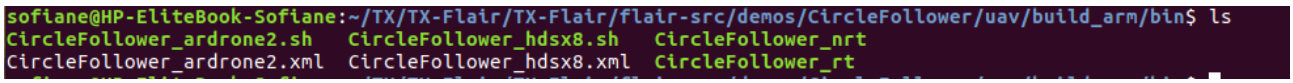


Illustration 59 : build_arm/bin files

The files that you have to load in the drone are:

- The executable file generated in your ARM folder (airdrone2 drones don't support real time so take nrt version).
- The launcher script file.
- The setup.xml file.

Create a repository where you will copy these files. To transfer the files in SSH we use SCP command:

```
$ $ scp path/to/source/folder root@drone_ip:path/to/destination/folder
```

The launcher script file:

```
$ scp $FLAIR_ROOT/flair-
src/demos/CircleFollower/uav/build_arm/bin/CircleFollower_ardrone2.sh
root@drone_ip:path/to/destination/folder
```

The executable file:

```
$ scp $FLAIR_ROOT/flair-
src/demos/CircleFollower/uav/build_arm/bin/CircleFollower_nrt
root@drone_ip:path/to/destination/folder
```

The xml file:

```
$ scp $FLAIR_ROOT/flair-
src/demos/CircleFollower/uav/build_arm/bin/CircleFollower_ardrone.xml
root@drone_ip:path/to/destination/folder
```

Configure Optitrack

Configure OptiTrack is necessary before starting drone flights.
Lauch OptiTrack on the dedicated computer, open a new project.
Now put the drone with its protection case in the flight area to be visible in the OptiTrack scene. It's important to put the UAV at the center of the room to avoid any collision with the walls during the flight.

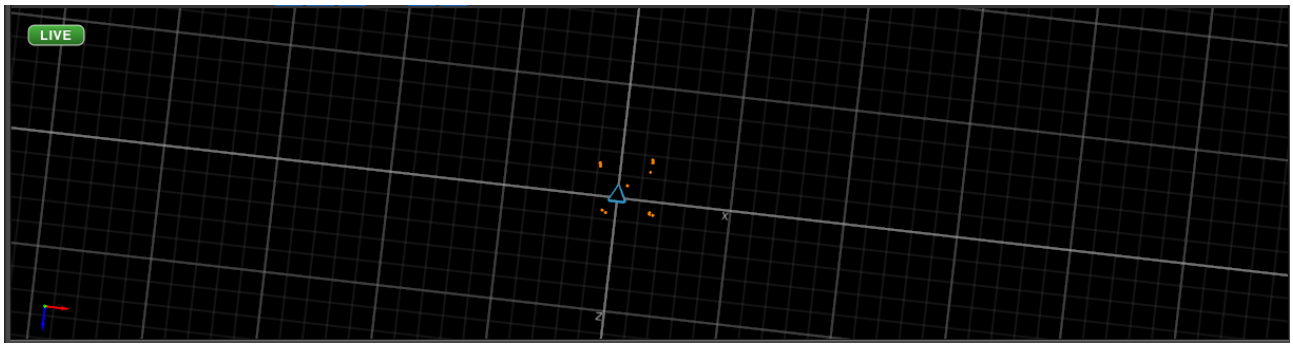


Illustration 60 : Drone markers in the Opti Track scene

After that, you should see in the OptiTrack software the points representing the drone markers.
Group all this points by selecting them and create an object, name it exactly like the drone name in your program launching script.

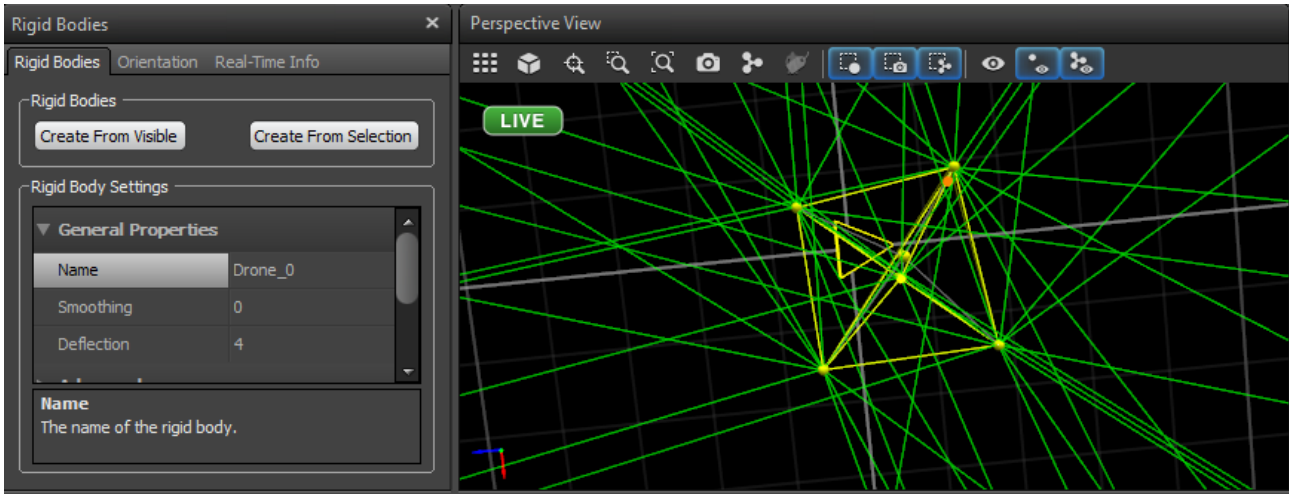


Illustration 61 : Rigid Bodies in OptiTrack

To represent the target we use an empty protective case, the markers in the case will represent the target, as you did with the drone, group the points of the target and name it with the same name as in your XML file.
Now the drone and the target are identifiable by OptiTrack.

How to know the uav and the target name?
You can find the names in your program script "CircleFollower_ardrone2.sh" for example.

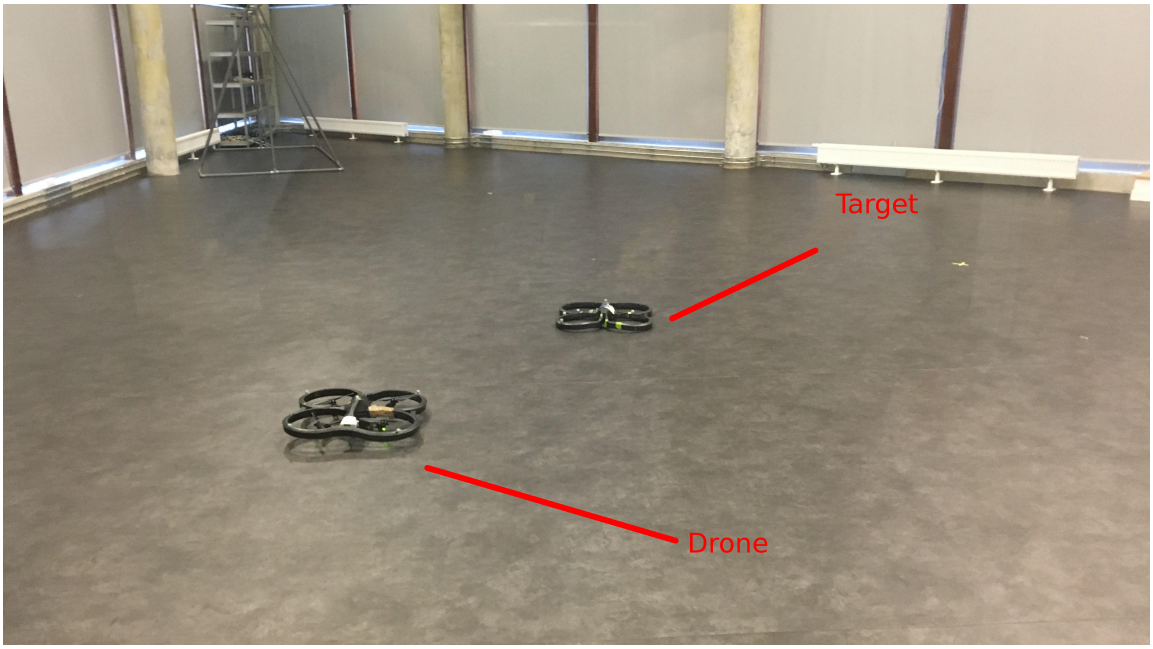


Illustration 62 : Drone and Target in the aviary

Ground station and script launching

Launch the flair ground station on the computer connected to the drone.

```
$ $FLAIR_ROOT/flair-bin/tools/scripts/launch_flairgcs.sh
```

Launch the dualshock3 script to connect your joystick in usb mode.

```
$ $FLAIR_ROOT/flair-bin/tools/scripts/dualshock3_local_usb.sh
```

Before launching the program script it's important to put the good parameters.

These parameters are : executable file , IP address of the ground station computer, XML file name and drone's name.

All the parameters can be changed in the script file:

```
$ $ vi CircleFollower_ardrone2.sh
```

```
./CircleFollower_nrt -a 192.168.147.193 -p 9000 -l /tmp -x CircleFollower_ardron
e2.xml -n Drone_0 -t ardrone2
~
~
```

Illustration 63 : CircleFollower_ardrone2.sh

Now your script is ready to be launched. Launch it on the terminal of the drone.

Press the Start button on the Dualshock3 to take off or land.
Press R1 and Circle buttons at the same time to start the circle trajectory.
To stop the trajectory and regain manual control of the drone press Cross button.
If you want to make an emergency stop and turn off the motors press Option button on the Dualshock3 controller.