Using FAUST with ROS
(version 0.0.09)

GRAME
Centre National de Création Musicale

December 2014
Contents

1 Introduction 5
  1.1 FAUST 5
    1.1.1 Design Principles 5
    1.1.2 Signal Processor Semantic 6
  1.2 ROS 6
    1.2.1 What is it? 6
    1.2.2 Concepts 7
  1.3 Using FAUST with ROS 9
  1.4 Audio Server 10
  1.5 Installation 10
    1.5.1 FAUST 10
    1.5.2 ROS 10
    1.5.3 Jack 10

2 Compiling a FAUST Program for ROS Use 11
  2.1 How does it work? 11
    2.1.1 Different steps 11
    2.1.2 With an example 12
  2.2 Compiling in a FAUST Archive 13
  2.3 Compiling in a Workspace 14
  2.4 Example 14

3 Using FAUST Nodes 15
  3.1 Run the Master 15
  3.2 Run a FAUST Node 16
  3.3 To Which Topics is a FAUST Node Subscribing? 16
  3.4 How to Escape from a Running Node? 18
CONTENTS

4 Metadata .............................................. 19
  4.1 DSP writing ........................................ 19
  4.2 Compilation ....................................... 20
  4.3 Run ................................................. 20

5 Common Error Messages ......................... 21
  5.1 The command does not output anything ............ 21
  5.2 No such file or directory ......................... 21
  5.3 [roslrun] error .................................... 21

6 Cheat Sheet ........................................ 23

7 Tutorials ........................................... 25
  7.1 DSP File ........................................... 25
  7.2 Compilation ....................................... 26
    7.2.1 If you installed FAUST on your machine ........ 26
    7.2.2 With FaustLive ................................. 26
    7.2.3 With the online compiler ....................... 26
  7.3 Run ................................................ 27
    7.3.1 Use ............................................ 27
  7.4 ROS tips .......................................... 28
    7.4.1 How to create a ROS workspace ? .............. 28
    7.4.2 How to make your workspace ? .................. 28
    7.4.3 How to source a workspace ? ................... 28
Chapter 1

Introduction

If you wanted to add some sound to your robot, or if you wanted to use your sound applications on a robot, it is now possible with FAUST and the `faust2ros` and `faust2rosgtk` commands.

FAUST (Functional Audio Stream) is a functional programming language specifically designed for real-time signal processing and synthesis. FAUST targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.

ROS (Robot Operating System) is a flexible framework for robot software writing. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

1.1 FAUST

1.1.1 Design Principles

Various principles have guided the design of FAUST:

- **FAUST** is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. FAUST is, as much as possible, free from implementation details.

- **FAUST** programs are fully compiled, not interpreted. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers.

- The generated code works at the sample level. It is therefore suited to implement low-level DSP functions like recursive filters. Moreover the code can be easily embedded. It is self-contained and doesn’t depend of any DSP library or runtime system. It has a very deterministic behavior and a constant memory footprint.
- The semantic of FAUST is simple and well defined. This is not just of academic interest. It allows the FAUST compiler to be semantically driven. Instead of compiling a program literally, it compiles the mathematical function it denotes. This feature is useful for example to promote components reuse while preserving optimal performance.

- FAUST is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: functional programming and algebraic block-diagrams. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a block-diagram algebra of five composition operations (; , ~ ◦ ◦). 

- Thanks to the notion of architecture, FAUST programs can be easily deployed on a large variety of audio platforms and plugin formats without any change to the FAUST code.

1.1.2 Signal Processor Semantic

A FAUST program describes a signal processor. The role of a signal processor is to transform a group of (possibly empty) input signals in order to produce a group of (possibly empty) output signals. Most audio equipments can be modeled as signal processors. They have audio inputs, audio outputs as well as control signals interfaced with sliders, knobs, vu-meters, etc...

For more informations about FAUST, please see faust-quick-reference.pdf and the tutorials in FAUST documentation.

1.2 ROS

1.2.1 What is it ?

Creating truly robust, general-purpose robot software is hard. From the robot’s perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own.

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

As a result, ROS was built from the ground up to encourage collaborative robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing
1.2. ROS

small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other’s work, as is described throughout this site.

1.2.2 Concepts

Filesystem level

The filesystem level concepts mainly cover ROS resources that you encounter on disk, such as:

- **Packages** are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.

- **Metapackages** are specialized Packages which only serve to represent a group of related other packages.

- **Services**: Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.

- **Messages**: Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS.

Computation Graph level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

- **Master**: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

- **Nodes**: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

- **Topics**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general,
publishers and subscribers are not aware of each others’ existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

- **The Parameter Server**: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.

- **Messages**: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structures).

![Figure 1.1: ROS Concepts in a Diagram](image)

### Names

Names are really important in ROS. Valid names have these characteristics:

- first character is an alpha character: `[a-zA-Z]`
- subsequent characters can be alphanumeric: `[a-zA-Z][0-9]`, underscores: `_`, or forward slash: `/`
- there is at most one forward slash: `/`

For more informations on ROS and tutorials, please have a look to the website: [www.wiki.ros.org](http://www.wiki.ros.org).
1.3 Using FAUST with ROS

The idea of using FAUST modules with ROS could be summed up in the following diagrams.

As shown on figure 1.2, the dsp file is compiled into a C++ file thanks to the FAUST compiler. Then, the C++ file can be compiled with catkin in a ROS package to create a ROS executable, that you can run with `rosrun`.

Once the executables coming from DSP files compiled, you can run and combine them with robotic applications (figure 1.3).
1.4 Audio Server

FAUST applications use the jack audio server. Make sure it is installed on your machine.

![APIs used by FAUST nodes](image)

1.5 Installation

1.5.1 FAUST

You can get FAUST on the FAUST website: faust.grame.fr. Either get it on the Source Forge Project, or browse the FAUST's git repository. You also can download the ubuntu package directly by typing: `sudo apt-get install faust` (probably an outdated version).

1.5.2 ROS

A ROS installation guide can be found on the ROS website.

1.5.3 Jack

A Jack installation layout can be found on Jack's github's page. You can download it as an Ubuntu package by typing: `sudo apt-get install jackd2`. 
Chapter 2

Compiling a FAUST Program for ROS Use

To compile a FAUST program for a ROS use, you can use either the faust2ros command, or the faust2rosgtk one, which adds a gtk graphic user interface to the simple faust2ros command. Note that all the FAUST compilation options remain.

Comment : The compilation can last between 10 and 20 seconds. This is completely normal! The bash script includes some ROS compilation with catkin_make.

BE CAREFUL! To run these commands, you need to have ROS installed on your machine. They are indeed using catkin_make and rosrund, which are ROS commands.

2.1 How does it work?

faust2ros (or faust2rosgtk) is a small command, which hides a lot of things. This section aims for a clarification.

2.1.1 Different steps

Step 1. Workspace Creation : if asked, a workspace can be created if it does not already exist. A workspace is considered as a ROS workspace if it contains a non-writable CMakeLists.txt in its src folder. If the user wants to get a .zip file, a temporary workspace is created.

Step 2. First compilation : the .dsp file is compiled a first time thanks to the FAUST compiler and the ros-callbacks.cpp architecture file. This step is dedicated to the ROS metadata. The C++ file is then compiled into an executable thanks to catkin, which uses CMake.

11
CHAPTER 2. COMPILING A FAUST PROGRAM FOR ROS USE

Step 3. Second compilation: the .dsp file is compiled again, thanks to the FAUST compiler and the jack-ros.cpp architecture file (respectively jack-gtk-ros.cpp). This step creates the ROS code, with a missing class. This class is then included with the execution of the previous executable, and the C++ file can be compiled with catkin.

Step 4. Cleaning: the package is compressed into a .zip archive and the temporary folder deleted.

2.1.2 With an example

Let’s imagine that I want to compile the harpe.dsp file. There are three parameters that can be changed: attenuation, hand and level. I want hand to be controlled through a topic called /my/robot/topic. This topic is going to deliver messages of type std_msgs/Int64.

My metadata declaration line will look like 

```
[ros:/my/robot/topic std_msgs Int64 data]
```

The second compilation (step 3 below in subsection 2.1.1) will add a class called RosCallbacks in which are:

- a ranging function, that re-scales values from the topic to fit with the FAUST sliders scale.
- a specific callback, called callback0, which sets the audio parameter value to the message value:

```c
void callback0(const std_msgs::Int64ConstPtr& msg,
               FAUSTFLOAT* zone)
```

- a subscribing function, which calls callback0 in the ROS way:

```c
void Subscribe(std::vector<FAUSTFLOAT*> zones) {
  ros::Subscriber* my_sub0 = new ros::Subscriber();
  *my_sub0 = nh_.subscribe<std_msgs::Int64>("my/robot/topic", 1, boost::bind(&RosCallbacks::callback0, this, _1, zones[0]));
}
```
2.2 Compiling in a FAUST Archive

In order to compile a DSP file into a FAUST archive, just type the command followed by your file:

```
faust2ros file.dsp
```

It should output:

```
file.zip;
```

and the resulting `file.zip` folder should contain a package called `file`, which contains a `.cpp` file corresponding to the DSP file.

If the DSP file is not in the current directory, make sure to type the right path. For instance:

```
faust2ros ~/faust/examples/myfile.dsp
```

Comments:

- If you want to use the `faust2rosgtk` command, the output will have a `.gtk` extension. For instance:

  ```
  faust2rosgtk file.dsp
  ```

  should output:

  ```
  file_gtk.zip;
  ```

- The zip file is located in the current directory.
2.3 Compiling in a Workspace

Thanks to the option `-install`, you have the possibility to create a package from your DSP file directly in a workspace you choose. Just type:

```
faust2ros -install faust_ws file.dsp
```

It should output:

```
file.cpp;
```

and you should have a faust_ws repository looking like this:

```
faust_ws
  __build
  __devel
  __src
    __file : File Package
      __include
      __src
        __file.cpp : File generated with the Faust compiler
        __CMakeLists.txt
        __package.xml
```

2.4 Example

Here is an example of a three files compilation.

Input:

```
faust2ros -install foo_ws -o foo1 file1.dsp
   -install foo_ws -o foo2 file2.dsp
   -install bar_ws -o bar file3.dsp
```

Output:

```
-foo_ws
  __foo1
  __foo2
bar_ws
  __bar
```
Once your DSP files are compiled into ROS executables, you can run them into a ROS master.

### 3.1 Run the Master

A **FAUST** node needs a master to run. You can check if a master is already running by typing:

```bash
rostopic list
```

Then, there are two possibilities:

- either you get the following message:

  ```bash
  ERROR: Unable to communicate with master!
  ```

  which means there is no master running

- or you get:

  ```bash
  /rosout
  /rosout_agg
  ```

  which means a master is already running.

To run a master, you have to type the following command:

```bash
roscore
```
3.2 Run a FAUST Node

Now that your master is running, you can run your FAUST node. It is quite simple. Type:

```
rosrun mynodepackage mynode
```

For instance, if your node name is `foo`, then type:

```
rosrun foo foo
```

If you get an error message looking like this:

```
[rosrun] Couldn't find executable named foo below /path/to/myworkspace/src/foo
```

then refer to section 5.3

3.3 To Which Topics is a FAUST Node Subscribing?

Once your FAUST node is running, it automatically subscribes to topics corresponding to the parameters you can modify, and to the widgets the gtk graphic interface has. For instance, if you use a FAUST node generated from the noise.dsp file (in the examples directory), the `noise_gtk` node will subscribe to the topic `noise_gtk/Volume` and the graphic interface will look like this:

![Figure 3.1: noise_gtk graphic interface](image_url)
3.3. TO WHICH TOPICS IS A FAUST NODE SUBSCRIBING?

A more complex example like the harpe.dsp file, which contains three widgets, can generate several topics to subscribe to:

```
/harpe_gtk/attenuation
/harpe_gtk/hand
/harpe_gtk/level
```

and the graphic interface can look like this:

![harpe_gtk graphic interface](image)

Figure 3.2: harpe_gtk graphic interface

If you want to change the topic name, just remap them while running your node:

```
rosrun myfaustpackage myfaustnode /topicname:=/newtopicname
```

For instance, to remap the `/harpe/hand` topic to `/play`, then run the harpe node like this:

```
rosrun harpe harpe /harpe/hand:=/play
```

**Comment:** The FAUST nodes subscribe to topics using std_msgs message types. Depending on the widgets you use, you can subscribe to default topics using either Float32 or Bool messages.

<table>
<thead>
<tr>
<th>Widget</th>
<th>Message type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>std_msgs/Bool</td>
</tr>
<tr>
<td>Check Button</td>
<td>std_msgs/Bool</td>
</tr>
<tr>
<td>Slider</td>
<td>std_msgs/Float32</td>
</tr>
<tr>
<td>Num. Entry</td>
<td>std_msgs/Float32</td>
</tr>
</tbody>
</table>
3.4 How to Escape from a Running Node?

To close a node running in ROS, you have two possibilities, depending on the graphic interface:

- If your node has a graphic interface, then quit by clicking on the red cross in the corner of the window.
- If your node does not have any graphic interface, then quit by typing Ctrl+C in the node's terminal window.
Chapter 4

Metadata

You might not want to use the float or bool standard messages for your topic, and compile the dsp file directly with the right topic name. In order to accomplish this, you can use ROS metadata.

4.1 DSP writing

It all starts in the widgets definition. Until now, maybe that you only wrote:

```
param = hslider("level", etc);
```

To use ROS metadata, you simply have to add square brackets with your topic parameters:

```
param = hslider("level [ros:/my/topic/name msg_type
msg_name field_name]", etc);
```

As an example, if you intend to use integers in a foo/bar/baz topic, you can type:

```
param = hslider("level [foo/bar/baz std_msgs Int32 data]
", etc);
```

And if you intend to use the y field of a Point32 geometry_msg, then type:

```
param = hslider("level [foo/bar/baz geometry_msgs
Point32 y]", etc);
```

You can also add the minimal and maximal values of the signal (4.1 and 9.3 for instance) by adding in the metadata declaration:

```
param = hslider("level [foo/bar/baz std_msgs Int32 data
4.1 9.3]", etc);
```

**BE CAREFUL!** The minimal and maximal values must be floats! Otherwise, compilation fails.
4.2 Compilation

To compile your dsp file, just do like you used to do before to find out this wonderful chapter about metadata: faust2ros or faust2rosgtk. It will add a RosCallbacks class in your C++ file, containing specific callbacks.

Comment: Even without any declared ROS metadata, a rosCallbacks class is created, but it does not contain any callback implementation. It is just an empty class.

You can then build your executable using catkin_make.

4.3 Run

To run your node, just do as usual, using rosrun or a launch file you wrote. The FAUST node creates two kinds of topics:

- Default topics, using Float32 and Bool standard messages
- Customized topics, created from ROS metadata.
Common Error Messages

Compiling can fail. Here are some common mistakes and how to solve them.

5.1 The command does not output anything

If, after typing your command followed by a file name, your terminal does not output anything like `myfile.zip`; or `myfile.cpp`; and returns only a blank line, make sure you are in the correct directory or you entered the correct path to reach the DSP file.

5.2 No such file or directory

If you used the `-install` option, make sure you typed the complete workspace path. For instance, instead of typing this:

```bash
faust2ros -install myworkspace ~/path/to/myfile.dsp
```

you should type:

```bash
faust2ros -install path/to/myworkspace ~/path/to/myfile.dsp
```

5.3 [rosrun] error

If, while trying to run a FAUST node (called `mynode`), an error message showed up saying:

```
[rosrun] Couldn't find executable named mynode below /path/to/myworkspace/src/mynode
```

Then you have to source your workspace:

```bash
source /path/to/myworkspace/setup.sh
```
• If your workspace is only a test workspace, then type:
  
  ```bash
  source path/to/myworkspace/devel/setup.bash
  ```
  
in your terminal.

• If your workspace is going to be your current ROS workspace, you can add it to the source directories:
  
  ```bash
  echo "source path/to/myworkspace/devel/setup.bash" >> ~/.bashrc
  ```
Chapter 6

Cheat Sheet

Compilation

- `faust2ros` compile a .dsp file
- `faust2rosgtk` compile a .dsp file with a GTK graphic interface

Options

- `-install` installation in a specified workspace
- `-o` rename the executable

Example

```
faust2ros -install ~/catkin_ws -o harpe4ros ~/dsp/harpe.dsp
```

Run

```
roslaunch harpe4ros harpe4ros
```

Metadata

```
[ros:/my/topic msg_type msg_name msg_field]
[ros:/my/topic msg_type msg_name msg_field min_value max_value]
```
Chapter 7

Tutorials

This tutorial will teach you how to use FAUST with ROS through the turtlesim package.

7.1 DSP File

Copy the following code into a text file and save it as roscillator.dsp.

```faust
declare name "roscillator";
declare version "1.0";
declare author "Grame";
declare license "BSD";
declare copyright "(c)GRAME 2009";

//-----------------------------------------------
// Sinusoidal Oscillator
//-----------------------------------------------
import("music.lib");

smooth(c) = *(1-c) : +~*(c);
vol = hslider("volume [unit:dB][ros:/turtle1/pose turtlesim Pose x 0.0 11.0]", 0, -96, -96, 0.1) :
db2linear : smooth(0.999) ;
freq = hslider("freq [unit:Hz][ros:/turtle1/pose turtlesim Pose y 0.0 11.0]", 1000, 20, 24000, 1);

process = vgroup("Oscillator", osc(freq) * vol);
```
7.2 Compilation

7.2.1 If you installed FAUST on your machine

In a terminal, type:

```
faust2rosgtk -install ~/catkin_ws ~/path/to/roscillator.dsp
```

The compilation can take between 10 and 15 seconds. Then, go in your workspace’s root and source the setup file (see section 7.4).

7.2.2 With FaustLive

FaustLive is available on Source Forge. Once FaustLive installed, launch it. Choose Open your File dsp and select roscillator dsp. The file will be compiled and executed. To export it, clic on Window/Export As… or press Ctrl+P. The Export Manager window should be opened.

![Figure 7.1: FaustLive Export Manager](image)

Select ros as platform, ros-jack-gtk as architecture, and binary.zip as binary. Then click on Export, and Save. Unzip your package in your current ROS workspace, and make your workspace with catkin_make, and source it (see the ROS tips in section 7.4).

7.2.3 With the online compiler

The online compiler is available on the FAUST Website. In Faust Code tab, drop your roscillator dsp file. In the Exec File tab, choose Ros ros-jack-gtk as architecture, and click on Download the executable file. Unzip your package in your current ROS workspace, make your workspace with catkin_make, and source it (see the ROS tips in section 7.4).
7.3 Run

In a terminal, run the master:

```bash
roscore
```

In two new terminals, run the turtlesim node and the teleop_key node:

```bash
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

In a fourth (and last) terminal, run your FAUST node:

```bash
rosrun roscillator_gtk roscillator_gtk
```

You should have four open terminals, a turtle window, and a FAUST window.

![Screenshot with the four open terminals](image)

Figure 7.2: Screenshot with the four open terminals

7.3.1 Use

To move your turtle, go back on the teleop_key terminal and use your arrow keys.
7.4 ROS tips

7.4.1 How to create a ROS workspace?

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

7.4.2 How to make your workspace?

```
cd ~/catkin_ws
catkin_make
```

7.4.3 How to source a workspace?

```
cd ~/catkin_ws
source devel/setup.bash
```